An extended library of collection classes for .NET

Niels Jørgen Kokholm
kokholm@itu.dk

Thesis,
Master of Science in IT, Software Development
IT University Copenhagen
March 2004

1 Introduction **1**

2 Background **2**

2.1 Collection concepts *2*

2.2 Data structures and algorithmic complexity *7*

2.3 .Net and The Common Language Infrastructure *10*

2.4 Other collection class libraries *15*

3 Problem statement **16**

3.1 Goal *16*

3.2 Quality/non-functional requirements *16*

3.3 Operation *16*

3.4 Revision *18*

3.5 Transition *19*

4 Methods **20**

4.1 Overall methodology *20*

4.2 Measuring performance *21*

4.3 Testing *22*

4.4 Development tools *22*

5 Modeling and Interface Design **24**

5.1 Item features *24*

5.2 Collection features *27*

5.3 Interface architecture *32*

5.4 Interface design (member selection) *35*

6 Class design and algorithmic design **39**

6.1 General implementation topics *39*

6.2 Collection classes *44*

6.3 Support classes *58*

7 Evaluation **62**

7.1 Functional test *62*

7.2 Performance tests *63*

7.3 Extensions, variations and missing things *66*

7.4 Platform anomalies *67*

8 Conclusion **68**

Literature **68**

1 Introduction

This is the report from the author's thesis project at the IT University performed to attain the degree of MSc in IT, software development.

The project was initiated by a suggestion of Peter Sestoft to further develop a library of collection classes for the .Net platform he had developed during a stay at Microsoft Research at Cambridge. The goal was to achieve a library of collection classes with a level of functionality like the library of the Java platform, where the standard .Net collection classes are much more limited in range. Peter was an active and inspiring advisor for the project, many thanks for that.

The reader is assumed familiar with the Java platform and object oriented development on such platforms. Knowledge of the .Net platform and in particular C# would be an advantage. We will present the main differences from the Java platform in the report. The reader is also assumed familiar with algorithms and data structures at the level of the first half of the standard textbook on algorithms by Cormen, Leiserson, Rivest and Shamir.

The library has been given the arbitrary name C5.

The report starts with some background material in chapter 2 on the concept of a collection, the .Net platform, data structures and a little on other collection class libraries. In chapter 3 we formulate a problem statement and a series of non-functional requirements, while chapter 4 describes the methods we have used. Then we describe the modeling of the interface architecture for the library in chapter 5 and the algorithmic and class design of the library in chapter 6. We evaluate the library in chapter 7 and conclude in chapter 8.

The complete source code of the library is presented in the first appendix. The next appendix is a short user's guide to the library together with a printed version of the interfaces and class reference manual. Finally, there is a third appendix with a list of remaining TODOs and a few example pages of code from the regression test suite and performance tests.

The library is included on a CD in source and compiled form together with a browsable user's guide and reference manual. Please see the Readme.htm file on the CD for a detailed description of the contents.

2 Background

We have sections on the concept of a collection from a non-implementation point of view, on data structures for use in implementations of collections as classes, on the platform the library will run on and on other similar libraries.

2.1 Collection concepts

In this discuss, we treat the concept of a collection and associated concepts from a conceptual point of view and not from a system or programming point of view, although it is of course with that perspective in mind. We do not try to analyze the concepts starting from first principles, but rather extract some common features of e.g. existing collection class libraries and data base literature. An important goal is to reach set of definite meanings to some important terms in order to make later narratives unambiguous and naming across the library more systematic and consistent.

2.1.1 Initial motivation

A collection is a collection **of** something. We will denote a constituent of a collection by the term "item". We only consider finite collections. Often one think of a collection as being built from items by adding or removing, but the items contained in a collection could also be given by other means, say some function. Many collections may report their items one by one – a process denoted "enumeration".

We could imagine collections of two levels of complexity: collections where all items play the same role and are of some common kind and collections where the items are of several kinds playing different roles. As examples of the latter, consider a collection of cities and the people living in them or of a graph with nodes and edges as items. One could use the terms homogenous vs. inhomogeneous to characterize the two kinds of collections, but we will only consider the former kind to be a true collection. The latter kind can of course be considered as a system of collections.

An important distinction of collections is whether they can contain several copies of an item or only one. This is the distinction between bags and sets of relational algebra. We will assume that all collections are either sets or bags (have bag respectively set semantics). Here we interpret bag in a strict sense: if a particular item is inserted twice in a bag, the bag will afterwards contain two copies of the item.

Note that the question of equality of items is of paramount importance in the distinction between bags and sets, but it also is important for understanding the notion of enumeration or the question if a given collection contains a particular item. In a purely mathematical setting, the question of collections and equality does not seem difficult, but it turns out to be a thorny issue in the setting of object oriented programming that is the real target of this project.

We also must consider the situation of collections with relations among the items. The simplest one is a sequence (-ordering) imposed either by the user or by a total order relation. This corresponds to an arrangement of the items in a linear graph. One could imagine relations corresponding to more general graph types, but that would take us outside proper collections and into graph structures, and we therefore limit ourselves to sequence ordering.

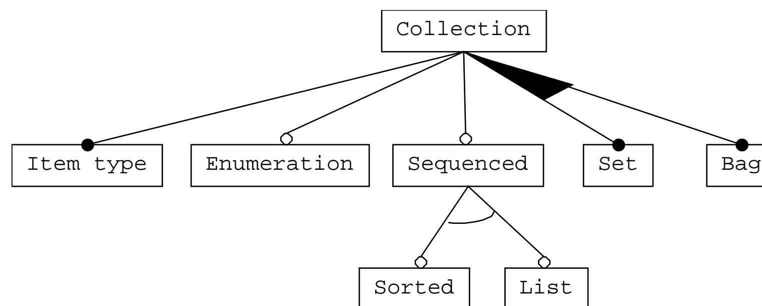
Note the ambiguity of the term “order”. In mathematics, this would usually refer to a total order relation, which corresponds to the term “comparer” in this report. In everyday language, order could easily refer to a sequence (-order) of the items, regardless of its origin. For this reason we try to avoid the term “order” by itself and only use it in combined terms like “sequence order”. Moreover, for a collection sequenced by a comparer, we use the term “sorted”, while if the sequence is chosen by the user, we use the term “list”.

Finally, since there is a clear connection between the mathematical concept of set and set collections, we should consider the meaning of the related mathematical concepts of relation and function in terms of collections. The interesting concept here is the analogue of a finite partial function, which in collection parlance will be called dictionary – a collection of pairs defining a partial function.

2.1.2 Feature overview

2.1.2.1 Collections features

The following diagram sums up the motivation above and the most fundamental features of collections. The notation is a *feature diagram* of Czarnecki and Eisenecker, [CE]. The meaning is that a collection unconditionally has an item type, may have enumerations and has precisely one of the two features set or bag. Moreover, the items of the collection may be sequenced and that may be by user listing, a sorting order or something else.



There are a number of additional feature areas:

- Which operations can we perform on the collection? See the next section.
- Can we (easily) access items of a sequenced collection by index?
- Is the collection mutable?
- Can we copy the collection? Can we access old versions?
- What possibilities are there for combining two collections?
- Is the collection a real collection, rather a result of a query on a collection or perhaps generated synthetically?

2.1.2.2 Operation features

This sub section tries to categorize the various kinds of operations one can imagine on collections: (the sub type names have been coined partly from SQL).

Main type	Sub type	Comments
Query		Operations on items that do not change the collection.
	Find	Search for an item in a collection or check for its presence.
	"Select"	Report items in a collection according to certain criteria.
	"Create View"	Like select, but the result is bound to the original collection.
	"Create Table"	Like report, but create a full-featured collection of the result.
Update operation		(May be single or aggregate, i.e. involve a single item or several items)
	Add	Add a certain item
	Remove	Remove a certain item
	Update	Replace an item with another one
	Retain	Remove all but certain items
Multi		Operations involving more collections, typically named join, merge, catenate, split etc.
Combined		Operations like updating an item or adding if the original was not found.
Other		Global Info/Diagnostic/About

2.1.3 Definitions

The purpose of this is to fix terminology. In a sense, this is an early design task and so it could belong to chapter 5. In fact, it is partly guided by criteria to be presented in chapter 3. However, since it largely is governed by and is an overview of common usage – we are not trying to give new definitions here, except fixing the usage of specific words – it is here to get this fix of terminology early. The sources of the terms are mainly the reference manuals of the .Net and Java collection classes and an article by Cook, [Cook]. To make such a list involves making many specific choices; arguments for some will be presented in later chapters.

In the definition table, we have marked the terms according to their usage as parts of program identifiers or in the narrative of this report of code comments:

Usage Code	Def
I	Interface
MP	Operation (method) name (or prefix)
MS	Operation (method) name suffix
N	For narrative use.
PP	Property name (or name part)

Term	UC	Definition	Caveats
Add	MP	Add one or more items as specified by a suffix or in the default way.	If collection is a set and item already there, do nothing and return false else return true.
Backwards	MP	Opposite enumeration order of original.	
Bag	N	A collection that allows several equal items inside at the same time.	If we insert some item twice, the collection will afterwards contain two copies of the item.
Clear	MP	Remove all items from the collection.	

Term	UC	Definition	Caveats
Client	N	The user of a class (a class itself)	
Collection	I	As a specific term in code, an enumerable with a built-in knowledge of its own size.	This definition to keep in line with the standard .Net library
collection	N	In connections like “collection class” any class considered as a collection in concept	More general than implementing ICollection<T>.
collection equal	N	The property of two items of being considered “the same” relative to the item hasher or item comparer of a specific collection.	Often just “equal”
Collection hasher	N	An item hasher for items of some collection type defined by the contents of the collections.	Can have sequenced or unsequenced meaning
Comparer	N	Cf. item comparer	
Count	PP	Size of collection in number of items.	
Dictionary	I	A map (finite partial function) that can be viewed as a collection of (key,value) pairs with the function property.	
Directed	I	A collection that has a defined enumeration order.	Supports two enumeration orders: forwards and backwards.
Duplicates	PP	Two or more equivalent items in a collection.	Used in NoDuplicates flag property
Entry	N	A (key, value) pair in a dictionary	
Enumerable	I	A collection that supports enumeration of its items. Two enumerations without intervening changes to the collection will deliver the items in the same order.	
Enumerator	I	An object that is the mediator of a single enumeration through a collection.	
External	N	Of an item hasher or comparer, the property of not being natural and hence “externally” defined	
Fixed Size	N	A dictionary, where the set of keys is fixed	Concept used in .Net, but not in this library
Hash Code	N	The result of applying the selected (or default) item hasher on an item.	Cf. item hasher
Hasher	N	Cf. item hasher and collection hasher	
Heap	N	The data structure literature notion of a data structure for priority queues based on, or similar in construction to, the classical binary heap.	There is also computational architecture and programming language notions of heap. For our platform, we will call it “the garbage collected heap”.
Index	N	An integer defining a specific position in a sequenced collection.	

Term	UC	Definition	Caveats
Indexed	I	A sequenced collection where the indices of items in the sequence order is maintained or readily available.	Some sequenced collection data structures do not allow efficient indexing.
Insert	MP	As prefix of operations on a list collection type: indicates that one or more items are to be added in a specific place.	No return value. Throws exception if collection is a set and item is already in the collection.
Interval	MS	Used with indexed collection types to denote the subcollection corresponding to a specific closed-open integer interval of indices	
Item	N	An object in the role of a (potential) member of a collection.	
Item Comparer	N	A binary function from two items to an integer defining an order relation on the items. Induces an equality binary predicate on items. Every sorted collection is based on one.	
Item Hasher	N	A binary equality predicate on items together with a compatible hash code function from the item type to integer. Every collection has one.	The hash code functions should for performance reasons separate items well, but this is not a formal requirement.
Item type	N	The kind of items a specific collection will hold.	
Key	N	An item of the primary side of a dictionary	
List	I	In indexed collection with a client defined sequence order.	
Natural	N	The property of an item hasher or comparer of being defined by the item type.	For hasher by the methods inherited from the object class, for comparer by implementing a comparable interface.
Persistence	N	The data structure literature notion	There is also an OO notion.
Predecessor	MS	The largest item less than a given one in some sorted collection.	This is the strong notion of predecessor.
Range	MP	The collection of items in a sorted collection that lies between certain item bounds.	
Read Only	N	A collection where update operations are illegal.	
Remove	MP	Remove an item from a collection.	
Retain	MP	Keep certain items in a collection while removing all others.	
Reverse	MP	Reverse the sequence order of a list, modifying the list.	Compare to backwards

Term	UC	Definition	Caveats
Sequenced	I	A collection whose items are arranged in a specific sequence order – by the client at insertion time or rearrangement time or by a comparer of the item type.	Also in combination sequenced (collection) hasher
Set	N	A collection that cannot contain two equivalent items.	
Sorted	I	A sequenced collection where the sequence (order) is defined by an item comparer.	
Successor	MS	The least item greater than a given one in some sorted collection.	This the strong notion of successor
Unsequenced	N	Comparing contents of collections and computing collection hash codes without respect to sequence order.	
Update	M	Replace an item in a collection with an equivalent one	But see "update operation"
Update operation	N	An operation that potentially changes the collection.	
Value	N	An item on the secondary side of a dictionary	May be used informally in narrative.
View	MS	A collection bound to (part of) another one with write-through updates.	

2.2 Data structures and algorithmic complexity

It is tempting to try to define a data structure as a set of algorithms cooperating to maintain certain data, but that would fit any class definition. We will be content with saying that a data structure is the item of study in the part of algorithmics studying data structures.

2.2.1 Computational models

In theoretical studies of the performance of algorithms and data structures, one bases estimations on one or more "computational models", in reality selected abstract machine models. There is a wide range of such models: cell probe, pointer machine, unit cost RAM model, IO model, cache oblivious model. While some are constructed to ease certain theoretical difficulties (in particular the cell probe model), others try to extract important performance characteristics from real machines in order to assist realistic predictions of the performance and in particular relative performance of algorithms. In statements on asymptotic completeness in this project, we will in principle base ourselves on a unit cost RAM model (unspecified details), but try to take the wide practical differences in access times among the levels in the memory hierarchy into account in a less formal way. The cost measures of the computational models will be something translating into a running time estimate – typically instructions count. Note that none of these models is made to ease the analysis of the effects of memory management by garbage collection.

2.2.2 Asymptotic complexity and practical performance

The theoretical analysis of algorithms is usually made in terms of asymptotic complexity statements in terms of the size, n, of the problem a certain operation will have a running time cost of O(f(n)) and a space use of O(g(n)) for some functions f and g. Instead of being upper bounds, bounds could

be lower bounds ($\Omega(f(n))$) or precise asymptotics ($\Theta(f(n))$). The bounds may also be of several kinds: worst-case, expected and/or amortized. For collections, the problem size n will usually be the size of the collection.

Expected comes in two senses (cf. [MR]):

- Expected (mean) cost over some random distribution of possible input data.
- Expected cost over some random choices made by the algorithm – irrespective of input data.

We prefer the latter, randomized, sense, where no specific bad input may give bad performance repeatedly – only with, typically very low, probability. In the table below, we denote an expected bound by an e attached to the O symbol. We always understand "expected" in the randomized sense unless explicitly stated otherwise

Amortized is a mean over a sequence of consecutive operations. An statement that some operation costs $O(f(n))$ amortized really means the statement that n consecutive operations costs $O(nf(n))$. In fact, the latter type of statement often is the only way to state amortized bound precisely.

The library will not be used for strict real-time applications and expected or amortized bounds will normally be fine as long as the worst-case bound is tolerable to incur occasionally.

When considering both running time and space use for some algorithmic problem it often is possible to make some algorithmic tradeoff between the costs. Similarly, when considering several operations to be supported in a data structure, one can usually make some tradeoffs between the performance of the various operations and no specific data structure will be optimal on every operation in isolation. Optimal data structures will then only be optimal in the sense that one cannot improve one operation without sacrificing the performance of another one (or the space cost). There is a third kind of tradeoff: between initial work to create a data structure and the cost of later operations. An obvious example is building indexes in a database table. This is most interesting when the later operations are all non-updating and we do not encounter it much in this project.

As is clear from the use of O notation, most theoretical studies ignore the "constant factors" in the cost behavior. This is because such a constant do not say anything about practical performance in itself, and because the constant factor typically is much harder to compute than the "exponent" of the algorithm. A notable exception is [Knuth], which contains many precise computations of the running times of algorithms on a precisely defined RAM model with specific instruction costs. Note that the lack of constants makes comparison of equal-asymptotics algorithms more difficult; in particular, there usually will be some possible space-time or multiple-operation tradeoffs that cannot be judged by asymptotic complexity.

The real question of interest is the practical cost of running an implementation of a data structure on actual hardware. The (theoretical) asymptotic complexity gives an incomplete, but very important answer or at least guidance. If two algorithms with asymptotic complexities e.g. $\Theta(1)$ respectively $\Theta(n)$ are implemented on a real platform, then for large enough problem size the former will be fastest; unless of course the practical constant factors are such that the crossover would happen above the capacity of the platform. Moreover, we could say that an $\Theta(1)$ algorithm should be efficient for all problem sizes unless the practical constants are humongous and an $\Theta(n)$ algorithm for, say performing a single update to a collection, will almost surely be unusable already for moderate problem sizes. One purpose of a collection class library is to provide implementations that are efficient for large problem sizes, since small problem sizes may be handled in a useful way even

with (theoretically) bad code. Therefore, we need to use data structures inside with first class asymptotic complexity, but also with small practical constant factors. The practical constant factors may be judged superficially from the seeming complexity of the data structure and more precisely from a detailed implementation design, but we need practical performance measurements to be on safe ground in our selection of data structure. The practical performance measurements will not be mere estimations of constant factors; they are bound to reveal imperfections of the computational model underlying the asymptotic complexity estimates. The practical results must be interpreted in terms of the expected asymptotic complexity and a more realistic view of the actual platform, but also in view of the possibility of a flawed implementation.

2.2.3 Data structures for collection classes

Here we will overview the most accessible data structures useful within general-purpose collection classes. Note that for these problems best-possible data structures are well known or the gap between the best known and the theoretical lower bound is negligible for our purposes of implementation. On the other hand, which of alternative good data structures perform best on .Net, and which detailed implementation choices are best, is less well known.

Some of the theoretically best performers are very complex and only marginally theoretically better than some much simpler ones so we will ignore the complex ones. A case in point is some the theoretically best predecessor structures in the form of search trees with adaptive degrees ([BF]).

The data structures can be divided into the following categories:

- Heaps
- Search trees
- Hash tables
- Linked lists
- Dynamic arrays
- Combinations of these

Heaps denote a group of data structures being based on the classical binary heap ([CLRS]) and implement priority queues perhaps with extra operations. The group includes ordinary binary heap, D-heaps, Leftist heaps, MinMax Heap, and Interval heap. There are also more complicated types with fast merging operations: binomial and Fibonacci heaps.

Search trees are tree formed data structures with items arranged (kept sorted) according to some item comparer. The interesting ones for collection classes keep the height of the tree low by some form of efficient auto balancing. Of the binary types, an early published type was AVL trees, simple and most well known is red-black trees and also important is splay trees that, in contrast to the two other ones, performs rebalancing on lookup operations too. Random treaps are binary trees kept balanced via randomization. Of higher degree trees, there are a,b trees, B-trees and a myriad of more exotic adaptive-degree trees with a little better asymptotics for membership and predecessor queries.

Hash tables utilize integer functions on the item domain to place items into internal, array-based tables and lookup items in those tables. The simplest types are called linear probing and linear chaining and have very good expected bounds. Perfect hashing ([CLRS]), while more complicated, achieves the same good performance in the worst-case sense for lookups.

Linked list structures come in multiple flavors as singly or doubly linked and with multiple items per node (blocked versions, e.g. for I/O efficiency).

Dynamic arrays are data structures, where the internal storage of items is handled by an array that are copied to and replaced by larger or smaller arrays as needed as the collection is modified. A variant type keeps the items sorted according to some item comparer.

The most interesting combinations are fast key lookup data structures like hash tables or search trees combined with a user-sequenced type as a linked list or dynamic array.

The following table shows an overview of the typical performance characteristics. All these results may be found in [CLRS]. All data structures we look at use O(n) space (unless we refrain from *shrinking* internal arrays for performance purposes).

Data structure	**Lookup By index**	**Lookup By value**	**Add**	**Insert**	**Remove**	**Remove by index**	**Remove by value**
Heap	-	-	O(logn)	-	O(logn)	-	-
Search tree	O(logn) Note-1	O(logn)	O(logn)	-	O(logn)	O(logn) Note-1	O(logn)
Hash table	-	$O(1)^{e}$ Note-3	$O(1)^{ea}$ Note-3	-	-	-	$O(1)^{e}$ Note-3
Linked list	O(n)	O(n)	O(1)	O(1)/O(n) Note-2	O(1)	O(1)/O(n) Note-2	O(1)/O(n) Note-2
Dynamic array	O(1)	O(n)	$O(1)^{a}$	O(n)	O(n)	O(n)	O(n)

^{e}: expected
^{a}: amortized
Notes:

1. If subtree sizes are maintained.
2. The O(n) cost is purely a lookup cost.
3. The expected bound is in the randomized sense (i.e. input independent) if the hash table uses a good randomized hash function internally.

There are quite efficient, but not very well known, persistent (cf. table in section 2.1.3) variants of several of these data structures. Implementing such persistent variants will enable strong functionality that is impossible to obtain in an efficient way based on the exported operations of the non-persistent variants. On the other hand, multiple collection operations like catenate may be harder to support in the persistent variants.

2.3 .Net and The Common Language Infrastructure

2.3.1 Overview

The common language infrastructure (CLI) is a standard from ECMA, the European Computer Manufacturers Association with the main components an object-oriented runtime system, the Common Language Runtime (CLR), and a comprehensive class library, [ECMA]. The CLI is constructed to support a wide range of programming languages and defines a particular one named C#. The rest of the report will use CLI instead of .Net when referring to the platform.

The main implementation of the CLI is in the Microsoft .Net SDK available for MS Windows platforms ([dotNet]). There is also a “shared source” implementation ([sscli]) released by Microsoft and open source/free versions under way from other organizations ([mono], [dotgnu]). The shared or open source implementations will run on MS Windows and (some) Unix type platforms.

The CLR is very close in appearance to the JVM supporting Sun MicroSystem’s Java programming language. Some main differences are some special features in CLR intended to aid support for several programming languages and some features for performance improvement – most notably genuine value types. The CLR philosophy is a little more relaxed than the Java philosophy, allowing redundant constructs to ease some programming tasks. The standard class libraries are similar in scope, but quite different in detail. The selection of collection classes in the standard library of the CLI is less comprehensive than in the standard Java library and consists mainly of dynamic arrays (as lists) and hash tables. The C# programming language is very close to Java.

For the practical study of this project, we have been using an alpha release of the MS .Net SDK, so some performance issues may be different in a production release. Note though, that the release can be considered well engineered and does not contain debug or trace code to slow it down to a crawl. Note that other implementations of the CLR will have different performance characteristics because of e.g. different garbage collection algorithms, different rules for inlining by the JIT compiler etc.

For the CLR, application code and library code is translated to intermediate language (IL or sometimes MSIL or CIL) and usually run via a JIT compiler, but can also be precompiled or (in principle) interpreted. Programs and libraries are divided hierarchically into namespaces; most of the standard library lives in the “System” namespace or sub namespaces of it. As an example, the standard collection classes lives inside the “System.Collections” namespace. Unlike Java, there is no forced correlation between namespaces and source code structure or deployment file structure.

Memory management on the CLR is done by garbage collection [Wilson]. The MS .Net SDK includes an advanced, generational mark and sweep garbage collector, which is self-tuning and adjusts generation sizes and collection frequencies to the actual allocation patterns seen.

We end by some specific features not present in current production versions of Java.

An important addition to the CLI that will appear in the next version is the introduction of generics/parametric polymorphism to the CLR and to the C# programming language. The C# syntax for using generic types is very similar to C++ templates, though less powerful: in C++ one can let code be conditional on the type of the generic parameters and have the conditional be resolved at compile (or link) time, while with CLR generics one cannot avoid some kind of runtime check to obtain that effect. Generics is important for making fast, safe collection class libraries. The following is a silly example of how the code for a generic class “Ten” with generic parameter “T”, a Push method to add items and an indexer to access items by index might look like in generic C#:

```
public class Ten<T>
{
    T[] a = new T[10];
    int ind = 0;
    public void Push(T item) { a[ind++] = item; }
    public T this[int i] { get { return a[i]; } }
}
```

The CLI standard include the Common Language Specification (CLS), identifying a certain subset of the legal CLR types and rules for using them, with the purpose of ensuring cross-language interoperation for language implementations that are "CLS-compliant". Unfortunately, generic classes are defined **not** to be CLS-compliant according to the prerelease manuals.

An innovative facility being added to the C# language is the so-called "iterator block". The following is an example of a method implemented with an iterator block as body returning an enumerator for an infinite synthetic collection consisting of the Fibonacci numbers (truncated):

```
public IEnumerator<int> GetEnumerator()
{
    int i = 1, j = 0;
    while (true)
    {
        int tmp = i; i += j; j = tmp;
        yield i;
    }
}
```

An iterator block is identified by having one or more "yield" statements, where control will be suspended and reiterated, and by being the body of a method returning one of a few specific classes. The C# compiler will create a few auxiliary classes, change local variables into fields and transform the body code to code for a suitable finite state machine in order to achieve the stated functionality. Iterator blocks greatly lighten the coding needed to implement enumerators for collection classes.

If a class, C, has a method, m, which is also defined with the same signature in, say, interfaces I1 and I2 implemented by C, then there will actually be three "vtable" entries for m in the IL code for C. Normally all the entries for m will point to the same code. but it is possible with so-called explicit interface implementation to have the entries point to different code. Then, if m is called on an object with runtime type C and compile time type I1, the I1 specific code will be called. The following example shows the use of this feature to make a poor man's covariance of return types:

```
public interface I1 { int m();}
public class C: I1 {
    int I1.m() { return 3; }
    public virtual double m() { return 5.0; }
    static void test() {
        C c = new C();
        I1 i1 = c;
        double r = c.m() + i1.m(); //8.0
    }
}
```

2.3.2 Performance issues

In this section, we will try to outline the special performance issues one faces when programming collection classes for the CLR in view of typical standard computational models like a RAM model with a memory hierarchy. The main point is operations or programming constructs that are "surprisingly" expensive on the CLR. We focus on cost in terms of running time and run time data size and not on JIT time or code size. A nice overview of these issues by G. Noriskin is [Noriskin].

2.3.2.1 Locality

Locality of memory references may be extremely important for the performance of code on modern machine architectures with several layers of memory caches with several orders of magnitude differences in access time. In the CLR, fields of objects or structs and elements of arrays are laid out consecutively in memory, so at that granularity we may control locality. Reference types are placed in memory without our control and if long-lived moved elsewhere by the garbage collector, so we may fear that accessing a reference field usually means a non-local memory reference and hence be expensive, although the way the garbage collector promotes objects could in fact lead to better locality. For linked (e.g. tree-based) data structures, using higher-degree nodes **may** mean less reference accesses for operations if relevant auxiliary information is kept in nodes.

2.3.2.2 Boxing (and unboxing) of value types

This is wrapping (and unwrapping) value types like integers or user defined structs inside objects allocated on the garbage-collected heap – a very expensive operation. The architecture of generics in CLR has been constructed to avoid boxing operations in code with a generic parameter that may have both value type and reference type instantiations. Note that in Java all "value types" are boxed unconditionally. When coding casually in C# one may easily overlook boxing done implicitly so one might want to check compiled IL code for boxing instructions. This has not been done during the course of this project though.

2.3.2.3 Value types vs. objects as building blocks of data structures

Value types will, unless implicitly boxed, be allocated on the stack or inline in arrays. Therefore, when we can avoid boxing, value type building blocks would put less pressure on the memory management system and may cater for better locality of data. On the other hand, large structs may be expensive in time and space to copy/duplicate compared to a reference.

2.3.2.4 Method calls

The CLI, unlike java, allows method calls to be declared as virtual or not. Both types are executed via a vtable entry and are of equal cost, except when the non-virtual call may be inlined. A method called through an interface also uses a vtable like table and is slightly more expensive (by a branch). With the typical (recommended) programming style of programming to interfaces not the concrete implementation classes, the compiled user code will contain calls to interfaces.

Some existing "System.Collection" classes (ArrayList) avoid making methods virtual in order to allow them to be inlined. While this may be of some value for a dynamic array implementation, it entails the problem for a collection class that either subclassing will be harder to do correctly or we must seal the collection class to disallow subclassing. Neither problem is acceptable for this library, so we shall make all public methods virtual unless special circumstances dictate otherwise.

2.3.2.5 Inlining or avoiding method calls

The JIT compiler will inline some method calls, when the call is not virtual, the target body is sufficiently small, only uses if-then-else control flow and satisfies some other conditions. Some rules have been outlined in [Noriskin], but the programmed rules are subject to change. The inlining rules are not very aggressive. Therefore, we should ourselves inline explicitly in the library for performance reasons, but only when the library code has been developed to a mature state.

2.3.2.6 Utilizing tail calls

The IL intermediate language of the CLR has a "tail." instruction prefix that instructs the JIT compiler to compile a tail-position method call to a jump eliminating the old activation record. The (current) C# compiler never emit the tail call instruction prefixes. Tail calls are not faster in themselves on CLR, but may leak less space by not retaining references from activation records. Using tail calls would mainly be beneficial for extensive use of (deep) recursion, which we do not

expect to do very much. Therefore we will not attempt to post-process the output from the C# compiler, inserting “tail.” instructions as allowed.

2.3.2.7 Object lifetime vs. memory management costs

Allocation of objects on the garbage-collected heap is (according to [Noriskin]) quite cheap compared to typical non-gc allocation schemes, because allocations in the first generation are linear (no free **list**). The expensive part of memory management is promoting objects to the second and later third (and final) generation and managing this process. Therefore, the recommendation in [Noriskin] is that object lifetimes should be either very short so such objects are never promoted or very long since the cost of promoting such objects can be amortized over a long period of time.

For a collection class built from some kind of cells containing one or more item, we cannot expect to be able to achieve this. Unless the collection class is built quickly and then kept static, we must assume that it will at all times have a varied mix of ages of cells.

2.3.2.8 Avoid many small objects

One way to reduce the memory management costs could be to try to avoid allocating many small and perhaps short-lived objects. Unfortunately, that is exactly what one would expect in an actively modified collection class built from cells. One remedy could be to make the cells larger – think higher degree trees instead of binary trees. Using value types where possible could help too. It is not advisable to maintain pools of cells, since such cells would tend to be old and make the garbage collection costs higher than if not maintaining the pool and using freshly allocated ones.

2.3.2.9 Write barrier check

The CLR uses a generational garbage collector. Whenever a reference field in object `o1` is updated, one (the CLR) has to check if `o1` is in an older generation than the reference being assigned to it and perhaps add that reference to the “remembered set” in order to be able to garbage collect the younger generation in isolation. According to [Gray] we should expect the barrier check to cost around seven machine cycles. Again, using tree nodes (or link nodes) with several references might avoid barrier checks by avoiding some updates to reference fields and updating bitmaps on deletions or restructurings.

2.3.2.10 Exceptions

It is very expensive to **throw** exception on the (MS implementations of) CLR, so exceptions should not be used for ordinary flow control.

2.3.2.11 Use of unmanaged or native code

Since the items we are collecting will often be reference types and so “managed” in CLI jargon, there does not seem to be much point in using unmanaged or native code performance wise – we would be crossing the line between the two worlds constantly, having to pin the items in memory and thus interfere with garbage collection. One might imagine that certain very specialized data structures on integers or strings could be more efficiently implemented completely in native code, but that is uninteresting for this project, where we are interested in generic collections.

2.3.2.12 Generics Code selection

As mentioned above, C# generics is less powerful than C++ templates when it comes to selection of code based on actual types of generic parameter instantiations. This might make it harder to make the best implementations for specific value types, say. We do not want a runtime check on item type on every access to select code. However, it may be interesting to do the check at generic parameter instantiation time or in a constructor and instantiate one of several auxiliary classes with the right type specific code in a field. Then the overhead at run time might be just an extra indirection.

2.4 Other collection class libraries

As noted in the introduction, the starting point of development in this project was the Generic Collection Classes library by Peter Sestoft with implementations of linked lists, dynamic arrays and red-black trees. This library is part of an effort to explore the use of the generic extensions of C# in various areas of programming ([PS]).

For the project of this report, the linked list implementation was copied from PS's code, but afterwards refactored heavily. Our tree and array implementations were in principle written from scratch, but with a keen eye on PS's code. Other aspects, of ideas, code organization and style have been inherited from that source, but we will not try to trace the connections further here.

We will very shortly describe some characteristics of other related collection class libraries.

The standard CLI collection classes of the platform contain both generic, non-generic and specialized collection classes with some non-uniformity among them. The base data structures are hash tables and dynamic arrays (as lists) with the variant of sorted arrays. This implies a serious lack of choices for the user when it comes to variability of performance characteristics. This seems to be a deliberate choice by Microsoft, leaving more extensive collection class implementation to third parties. The actual collection classes often have much more methods than the collection interfaces they implement, so those interfaces can be considered lightweight. Whenever reasonable we will follow these classes with respect to naming of classes and methods.

The Java standard collection classes of the current production version are based on a more comprehensive range of data structures: linked lists, dynamic arrays, linear hash tables, and red-black trees. The interfaces are more heavyweight than those of CLI, but there is the peculiar policy that it is allowable or even recommended for a class to **not** implement a method of an implemented interface honestly if the resulting method would be very slow. Instead, the implementation just throws a NotImplementedException! We will take Java as the main inspiration for the comprehensiveness level and general architecture, but not follow the idea of dishonest interface implementation. There exists a commercial collection class library for Java, JGL ([JGL]) much inspired from C++ STL, see below.

Smalltalk-80 also has a comprehensive set of collection classes. These are discussed with suggested improvements by William Cook ([Cook]). We have been inspired by his insistence on consistent naming, separation of inheritance hierarchy from interface hierarchy and by some of the Smalltalk types, like Updatable, Sequencable, Indexed.

The Standard Template Library (STL) of C++ contains a number of collection classes, but the library is organized in a different way then the one we are tageting, e.g. no use of interfaces. The people working on STL are very concerned with efficient algorithms and the sorting algorithm we have chosen to employ in our library ([Musser]) has grown out of work on implementing STL.

3 Problem statement

The problem to be attacked in this work will be stated as a top-level goal, the real problem statement. Then we will present some quality requirements for guiding the implementation.

3.1 Goal

Create a library of collection classes for the CLI that can assist expert and non-expert programmers on the platform to develop correct and efficient applications.

The library should at least fill the gaps in the standard "System.Collections.Generics" namespace compared to standard collection class libraries for related object oriented languages.

The library must utilize the generic extensions to the CLI announced for version 2.0 of the MS .Net SDK (fall 2004) – available in alpha release form since August 2003– and scheduled for an upcoming revision of the ECMA CLI specification.

3.1.1 Comments

The related languages would be in particular Java and also Smalltalk and C++.

Experts are the ones who know data structures and could implement them given time – non-experts would be the ordinary application programmer with little knowledge of algorithms and complexity and a propensity to use arrays (or linked lists) for everything.

The introduction of generics promises both improvements in the areas of safety because of more precise typing and of performance because of the elimination of type casts, checks and boxing. Nowhere would these improvements be more obvious/visible than in with collection classes.

Many data structures are not generic but specialized for one or only a few item types, typically strings or integer types, and can be much more efficient in their domain. A library of collection classes should implement the most generally useful ones of these, but we have chosen to concentrate on generic data structures.

3.2 Quality/non-functional requirements

We want, of course, the design and implementation to be of a high quality. In the rest of the chapter, we discuss which quality requirements are particularly important. The discussion will be structured according to the headings of McCall and Matsumoto with some additions from ISO 9126 through the presentation in Lauesen [Lauesen]. Note that the users are the application programmers.

3.3 Operation

3.3.1 Integrity (security)

The library itself will not have any authentication or access control features, but it might be called by code running with restricted CLI security privileges, say untrusted user code downloaded from the client to a web server. The library should not allow uncontrolled elevation of the privileges of the untrusted code.

3.3.2 Correctness and Reliability

It is very important that the user can feel safe that the classes will work according to specification under all circumstances. There are several points to this: using mature, proven data structures inside the collection classes; using mature coding techniques; and using effective testing methods.

There is a special issue of fault tolerance: the library must throw exceptions when relevant and not silently ignore errors. Internal data structures should not be compromised by erroneous or illegal input from user code.

Another aspect of fault tolerance would be what happens when a user-supplied operation like a comparer or hasher throws an exception. It must be defined as a breach of contract by the user to supply such an operation to the collection class, and the collection class has no way of making sure the supplied operation will not throw exceptions. It is not reasonable that performance (or code readability) suffer seriously by guarding calls to user-supplied operations with exception handling blocks; moreover, it is not obvious what such a handler should do. Therefore, it is up to the user to make sure the contract is upheld and so in the case of violation the collection class may be left in an inconsistent internal state.

3.3.3 Usability

The library should take away or at least lighten the burden on programmers of employing good efficient algorithms within the area of collections. Thus, it should implement simple data structures to lift the burden of tedious (and so perhaps error-prone) programming. It should also implement complicated data structures that might otherwise be too difficult to employ. Moreover, complex variants of the internals of such data structures that at little runtime cost can support extra collection class functionality should be considered positively.

In the absence of user surveys, we should use mature interface architecture and naming of types and operations should reuse good patterns from similar libraries.

The library should fit the needs of the programmers. We propose three success criteria for type and operation selection:

- Functionality and performance is intuitive i.e. non-surprising for an educated user.
- It is difficult for clients to achieve similar performance and/or functionality without.
- Convenient to use, supports common good programming idioms

The middle point is not obligatory. Naming is an important point in the first and last points. Emphasis should be put on the naming of types and operations to consolidate:

- Tradition in object oriented languages (for programmers with such knowledge)
- Tradition in the theory of efficient data structures (for programmers with such knowledge)
- Consistent use of names or parts of composite names
- Use of suggestive and not overly long names while avoiding ambiguity.

Keep the complexity of the system of types – in particular interfaces – low to assist the user's overview. The number of operations in interfaces should be comprehensive to permit substitution of one collection class for another one when there are several relevant classes with different performance characteristics. Thus, we should have a minimum of operations in collection classes not in their implemented interfaces. This goal is contrary to the current CLI collection classes,

where the interfaces are much more lightweight than the classes, and comes at the cost of making it harder for the user to implement the interfaces.

We want to allow the user to be able to program to the interface not to a specific implementing class, so the classes must support all published operations, even if a specific operation is difficult to implement in a specific data structure. This is contrary to some Java collection classes, where some interface operations are implemented in earnest – the operations simply throw an exception.

Conceptually orthogonal features should be independently selectable, but only to the extent that the complexity of the interface architecture and the data structures available allow it. We must avoid a combinatorial explosion in the number of interfaces.

3.3.3.1 Documentation

The library documentation is of primary importance for the usability. The implementation must be accompanied by comprehensive usage and reference documentation organized in a way that will be familiar to users. Emphasis should be put on guiding the user in selecting the best classes for a particular job, e.g. by documented performance comparisons among the implemented classes. The documentation must clearly explain the performance characteristics of the library classes. Contents should include Overview, Guide to selection, class/member details. In particular, the documentation should clearly warn about which of the operations on a specific class are particularly expensive and thus should be used only sparingly.

3.3.4 Efficiency

The collection class implementations should use first-class base data structures or algorithms – having good asymptotic complexity and good constants. Ideally, the very best data structures of a certain kind and perhaps some simpler runners up should be implemented and their practical performance compared to choose the one(s) best suited for this platform. In practice tradition, time constraints and gut feeling on behalf of the implementers of the library will be deciding factors on the data structures to be evaluated. Theoretical complexity/efficiency predictions should be backed by documented experimental evidence, as should choices made between design alternatives.

Complexity in itself of implementation of a data structure should not be an argument against its use in a collection class – on the contrary, complexity could be an argument for implementing it in the library if there is useful collection functionality to be gained. Since there will rarely be a single data structure whose performance is the best for every operation individually, we should employ a comprehensive set of algorithms/data structures with a wide spectrum of performance fingerprints. Efficiency for large collections is more important then utmost efficiency for tiny collections.

3.4 Revision

3.4.1 Maintainability

To make it easy to localize and safely repair errors we will try to follow good coding practices for naming, indentation etc. We will use the naming guidelines appendix to the C# standard. MS does not seem to have published a version for this .Net SDK of their fxcop code-checking program, so the control of conformance to coding standards will be manual. The source code will be organized into directories according to base data structure type and large classes split out to individual source files (no formal rules for the splitting).

Documentation of the internals will be supplied by this report.

Another role for maintainability will be played by the existence of comprehensive automatic regression test suites that can easily be extended to check that code changes do not break functionality.

3.4.2 Testability

The requirement is that the automatic test suites should be easily extensible so that one can test new functionality together with checking that old functionality is not broken.

3.4.3 Flexibility

How easy is it to expand the library with new features? We should try to avoid making it hard to implement the interfaces or subclass the collection classes. We will be making the interfaces heavy. That on the one hand makes them harder to implement, and on the other hand, makes them strong machine checked design guidelines.

3.5 Transition

3.5.1 Portability

The implementation should only use platform features (expected to be) standard, not use native code and not use "unmanaged" code.

For reasons of availability of development tools, the library will only be tested on MS .Net SDK (PDC alpha). This will mean that specific performance features of this platform – including possible performance quirks of an alpha implementation – may influence detailed design decision, but that seems unavoidable.

3.5.2 Interoperability

Since the CLI platform is designed to support many programming languages, it would be nice if the library was language neutral, but we will only try it from C#. The preferred way of assuring language neutrality on the platform is to be CLS compliant, but the documentation of the platform version, we use states that generic types are **not** CLS compliant.

3.5.3 Reusability

This refers to the question of reusing the library source code in other projects. We intend to publish the library with a BSD or MIT/X11 style open source license. This is the license the Mono project prefers for add-on libraries and will be more palatable for Microsoft than, say, a (L)GPL license.

4 Methods

4.1 Overall methodology

To create a library of collection classes for the CLR, we could envision employing Domain Engineering in the sense of [CE], develop a new one with more lightweight methods or by porting existing code from java to C#. The first one is not an option because of the resources needed for user surveys, comparative study and evaluation of actual usage patterns for collection classes and data structures. As for the last idea, we are not aware of any generic collection class libraries for Java that could be used.

Therefore, we will use modern lightweight object oriented system development methods. Instead of the initial extensive exploratory steps of domain engineering, we will just rely on the analysis of other library implementers as a reference point and as mentioned earlier some preexisting C# code.

One major issue in the chosen approach is that unlike the typical business application setting targeted in most literature on systems development, we cannot do conceptual modeling only in problem space. We need to analyze solution space concurrently because that is the only way to categorize which problems can be solved, in the sense of which purely conceptual collection classes (with specified performance characteristics) may actually be implemented.

The following is a list of tasks to perform. The order given is a normal logical progression, but in practice, there will a lot of overlap/iteration.

- Set goals.
- Study existing libraries and if possible published design rationales.
- Conceptual modeling (what **is** a collection)
- Study data structure literature and overview data structures (as bases of collection classes)
- Establish guiding principles for library architecture and type design
- Design interfaces.
- Design classes/algorithmic design.
- Implement, measure, test and document the classes.
- Evaluate.

While we try to describe the modeling and design in a logical progression it must be stressed that the library is designed as a whole, where the work on implementation details may inspire the “early” design principles.

The goals for this project have been described in chapter 3. The next four points have been treated in chapter 2. The interface design will be established in chapter 5 and the implementation will be described in chapter 6.

Whenever we mention something that would be interesting but has not been implemented, the reason is time constraints, which is rarely stated explicitly. We have tried to prioritize so that the most important features were finished first. Without a precise plan up front there necessarily will be mistakes in this as will be clear from the TODO list in the appendix.

Below we will describe a few more details on measuring and testing and on the development tools used.

4.2 Measuring performance

Our performance measurements have been concentrated on synthetic micro benchmarks, i.e. benchmarks consisting of a few operations in a loop that is repeated many times and timed in order to assess single-operation times. We have used them to

- Compare performance of implementation alternatives, both for choosing the implementation of a specific feature and for deciding whether some sub feature is worth including.
- Check the accuracy of the asymptotic complexity for predicting performance (also a check of the correctness of the implementation).
- Compare different classes to gauge their useful range.
- Compare to other similar classes to gauge the quality of this implementation.

In practice, only the first point has been done as much as one could wish. Details of some results will be presented in chapter 7.

We want to measure the performance of the collection classes as a function of the size of the collection. The most used type has been “full saw tooth”, where we alternately add items until a prescribed maximum size and remove items until empty and repeat until some large total number of operations is reached. We repeat for varying maximum size. Since half the operations will happen at a size at least half the maximum size, this should give a fair view of the dependence of operation speed on the size, but note that we only get a mean of the add and remove operation times. Depending on the precise class, we would use random add and remove or a systematic procedure like LIFO or FIFO. We should also measure lookups. The tests should be performed both with small value type items (like integer), large value type items and reference type items, but in practice we have almost only done the first type.

Garbage collections tend to use time in bursts, so we should let the individual benchmark timed loops run so long that most of the memory management costs associated with each measured time is included in that time. During initial tests with a smaller number of iterations, one often sees systematic artifacts on the size-to-time graphs seemingly due to garbage collection. The timing of the loops can use a processor time counter or elapsed time counter. Our graphs are based on the former that seems more reproducible, but we have always measured both for control.

Micro benchmarks may be misleading by exaggerating certain platform features. Perhaps a micro benchmark happen to use the memory hierarchy very efficiently, while normal usage of the same collection class would not have such good locality of memory access because of other code. Another source of problems could be that the type of data used (random) or systematic is not representative of the real environment in which the classes will be used.

There has been a single more full-application oriented performance test to compare various tree persistence alternatives, cf. chapter 7.

We are not aware of any standard benchmark suites to measure the performance of collection class libraries like the present.

Microsoft recommends using a profiler when tuning the performance of code on the CLR. We are not aware of any profilers readily available for the version of the platform we use.

4.3 Testing

The main testing of the library has been done via a regression test suite developed side by side with the library. We have used the open source NUnit tool ([NUnit]) to assists in organizing and running the tests. We have typically run the tests very often, sometimes after every compilation.

The design of the test cases has mainly been as black-box test in the usual "where could we expect something to go wrong"/corner case style. Nevertheless, some tests have been made systematically designed to cover all corners of code in the case of trees and hash tables.

We have used a custom code attribute, Tested, to mark which public methods of classes is deemed sufficiently covered by test cases.

As a supplement to the unit tests, we have performed tests that performed large numbers of random add and remove operations with an integrity check (Check() method) in between each operation.

At some points, we have put Debug.Assert(…) statements in the code to check invariants, but nothing systematic here.

4.4 Development tools

The alpha release of the .Net platform we have used contains Microsoft Visual Studio.Net (VS.Net), supporting C# development and other languages, and the .Net SDK with command line tools like the C# compiler, csc, debuggers and nmake. This implies a choice of the alternatives: use VS.Net or us a suitable program editor (emacs) together with the command line tools. The first alternative was chosen because the author was very familiar with the second one.

4.4.1 VS.Net

VS.Net is an integrated development environment with a language sensitive editor, debugger and build management. We have used features as context sensitive completion extensively. A nice feature is that if one writes the directive that some class implements some interface, the environment will offer to create dummy implementations of all members of the interface not already implemented. VS.Net puts a special interpretation to comments like "//TODO: …", whose text will be dynamically maintained in a TODO list. During development, this was very convenient, because you get the best of both worlds: the comment at the right place and an overview.

We have used the outlining features and organized the source code in outlining regions normally in the following order: fields, nested classes, private methods, constructors followed by one region for every interface implemented (directly or indirectly).

We have used conditional compilation by defining preprocessor symbols a lot for being able to compare implementation alternatives. Typically with as much code sharing as possible between alternatives, which unfortunately tends to make the code less readable than if one e.g. just used the symbols to select whole method implementations. Most of the preprocessor symbols should be removed, since a production version should choose among the alternatives, but for the tree implementation a symbol choosing between set and bag semantics will survive.

C# puts a special meaning to comments starting with three slashes, ///, as class documentation and when given the /doc flag, the compiler will extract the comments to a flat XML file keyed by a mangled class or member name. We have used this to generate the reference manual, see below.

One organizes a "solution" in projects, which will usually create exactly one CLR assembly file each. We have made a project for the library itself, a project for the regression test suite, a couple of projects for performance test and special tests, a project for documentation extraction and a project for extraction of "Tested" attributes to a database.

4.4.2 NUnit

This is a toolkit to assist unit testing C# programs, in particular for automatic regression testing. Test cases are defined by custom code attributes, best illustrated by an example:

```
using NUnit.Framework;
[TestFixture]
public class Inserting {
    private IList<int> lst;
    [SetUp]
    public void Init() { lst = new ArrayList<int>(); }
    [Test]
    public void Insert() {
        lst.Insert(0, 5);
        Assert.AreEqual(5,lst[0]));
    }
    [Test]
    public void IsEmpty() {
        Assert.IsTrue(lst.IsEmpty));
        lst.Insert(0, 5);
        Assert.IsFalse(lst.IsEmpty));
    }
}
```

The [SetUp] code will be run before each individual [Test] case. There is [TearDown] to cleanup after each [Test] case and [ExpectedException(…)] to test throwing of exceptions. We have used the GUI test runner, which shows the test cases in a tree built from namespaces and [TestFixture]. After a test run, good test cases become green, bad red. One may select parts of the tree to run.

4.4.3 Other tools

We have used CVS for version control (and backup).

A small Perl script (set2bag.pl) generates tree bag code from tree set code. To be run manually after modifications to the tree code.

An extension of docNet, a tool supplied as source code by Antonio Cisternino. The tool generates an XML file of type and member descriptions via reflection. We extended it mainly to support generics. The output is consolidated by XSLT (msxsl.exe) with the XML file of documentation comments produced by the compiler to produce the html reference manual.

We developed a tool in C# to export various custom code attributes defined by the library to an MS Access database used for analyzing the data. In the end, this was only used for tracking [Tested] attributes.

5 Modeling and Interface Design

In this chapter, we shall describe how the collection concepts of Chapter 2 are modeled in the library taking into account the available data structures.

During the chapter, we will formulate general principles as guides for the modeling in order to make the final design uniform. In reality, the principles were developed side by side with other work on the library.

We will first discuss modeling features of single items and features of collections. Then we discuss the interface architecture of collections and dictionaries. Finally, we discuss the design of the interfaces in terms of selection and naming of their members.

5.1 Item features

Here we treat the subjects of item equality, hash codes and comparers. We also discuss collections as items of other collections and touch on items whose identity is decided by a subpart.

5.1.1 Equality

The most fundamental feature of items to model in object oriented terms is their identity, i.e. how can we say that an item "is" in a collection or that an item returned from a query operation on a collection is "the same" as some "other" item?

Items of a particular collection will be modeled by objects of the item type – reference or value – of the collection. We need to be able to decide when two such objects are "equal". There are two canonical meanings of that in the CLR: one given by the method Equals inherited from System.Object and perhaps overridden and reference equality given by the ReferenceEquals method of System.Object. Unfortunately, these meanings are too restricted. The items will normally be defined independently of the collection and will not necessarily have an Equals method that fits the needs of a specific collection. A type may have several different meanings of equality relevant for collections of items of that type; see e.g. the question of collections as items treated below.

For these reasons, every collection class will have its own distinguished meaning of item equality given by an item hasher or an item comparer, according to the nature of the internal data structure.

5.1.2 Hashers

When the internal data structure does not identify items via a sorting order, the equality will be given by an item hasher object, defined as implementing the generic IHasher<T> interface:

```
interface IHasher<T>
{
    int GetHashCode(T item);
    bool Equals(T i1, T i2);
}
```

Here T is the generic parameter to be instantiated by the item type. (Here and in following code snippets from the library, we have elided access modifiers like "public" for easier readability.)

The methods should be consistent so that equal items have equal hash codes. They must stable so that repeated calls on the same items (in binary sense) yields the same results or implementations must at least state precise conditions under which this will happen so that the user may know if the hasher with certainty can be used safely as an item hasher in a particular setting. The methods must never throw exceptions.

The hash codes, the results of calls to GetHashCode, are not required to have any particular properties of distribution or independence. An implementation should try to make non-equal items have different hash codes with high probability.

Every item type has a natural hasher defined by the methods inherited from System.Object. In the next chapter, we describe helper classes for creating natural hashers and various external hashers.

Note that our IHasher<T> is very similar to System.Collections.Generic.IKeyComparer<T>, but we have made our own definition to avoid the use of the word "compare" in a situation where we do not talk about a sorting order.

5.1.3 Comparers

For internal data structures based on comparing items (a sorting order), the equality of items will be given by an object of the IComparer<T> class:

```
public interface IComparer<T>
{
   int Compare(T a, T b);
}
```

We interpret the result of Compare in the usual way, that "a" less than, equal to respectively greater than "b" if the result is negative, zero respectively positive. The Compare method must be consistent, defining a sorting order. Like a hasher, it must also be stable and not throw exceptions.

If an item type T is comparable in the sense that it implements one of the System.IComparable or the generic IComparable<T> interfaces, there is a natural comparer for that type. The next chapter will describe some helper classes for creating such natural and some external comparers.

There may be cases, where the natural comparer of a collection identifies items too easily. Consider a priority queue, where priorities are given by an "int priority()" method on the item type returning only a few different values. If we use a tree data structure for the priority queue, we cannot just create a comparer by comparing priorities, but must use extra properties of the items to distinguish them in the comparer so they will not be identified inside the tree. On the other hand, if we use a binary heap data structure, then it will be safe just to use the priorities to define a comparer since the heap will not identify items inside. Implementations of collection classes based on item comparers should document whether it is safe to use a "weak" comparer like this in this way.

There may also be cases, where one wants to use a comparer that strictly speaking is not perfectly self-consistent. Consider a geometric problem where we want to compare coordinates of points in the plane. If the points are given by floating point coordinates with possible round-off errors it is tempting to use a modified standard comparer of double values that identify two numbers if they are less than some small numbers. If we can assure that no false coincidences will happen, such a comparer implementation would be fine. We will use it in a geometric example in chapter 7.

Note that it is possible that a comparer (or a hasher for that matter) could have some internal state influencing its results. This is the case in the above-mentioned geometric example, where we compare y-coordinates of line segments at a specific, but varying, x-coordinate. If no line segments cross in an area of interest, such a comparer can be used as basis for a collection object.

5.1.4 Embedded keys

Sometimes we want to identify or compare items by just some "key" part, because the rest of the data in the item depends on that part. We considered having an "IKeyer<T,K>" interface that would define a method extracting the key and which together with a hasher or comparer for K would induce a hasher or comparer for T, but we found no real use for it.

A special case of embedded keys is dictionary entries as (key, value) pairs. Inside the dictionary, we would like to identify such pairs by their first component as an embedded key. The library defines an entry type (a value type) "KeyValuePair<K,V>" and hashers and comparers for it based on looking at keys only.

5.1.5 Collections as items

The collection objects we create may themselves become items in collections, and while it is possible that the user would want reference equality to identify collections, it is much more likely, that equality based on contents is the relevant one. Unfortunately, for collections with a sequence order, equality based on content equality may be either defined by comparing items one by one in sequence order or by comparing contents irrespective of sequence order – we cannot assume the user would always want to use the sequenced way to compare contents. Because of this ambiguity, we cannot just override "object.Equals" on collection classes to define contents equality. We could define an equality method (override) that would choose one or the other equality concept based on a flag property, but that would preclude that a given collection could simultaneously be item in a collection with sequence order based equality model on the items and another one with unsequenced equality model (the idea is ugly anyway). Thus, we are left with implementing the two equality concepts via individual methods as described below.

By unsequenced equality of contents, we will always understand the "with multiplicity" interpretation – known from relational algebra as "bag" equality.

To define hashers with these two equality concepts we will need to define corresponding hash code functions. These functions may only depend on the contents, one not even on the sequence order, and must be universal across collection classes.

For the unsequenced hash code, we must use the hash codes, c_1, c_2… of the items and are in practice restricted to a collection hasher of the form $h(c_1)\square h(c_2) \square\ldots\square h(c_n)$, where h is some integer function and $\square$ is some associative, commutative integer operation. Of the simple choices for $\square$ only bit wise xor and addition does not suffer from values that restrict the result and xor is terrible for bag collections (an even number of some item would be ignored). Thus, we should select addition, but then we need some good hash function for h, else the collection hasher would be bad for an item type like integer with hash codes equal to the items themselves. Unfortunately, the current library uses an unsequenced collection hasher that just uses xor to sum the hash codes of the items with no intermediate hash function, then adds the size shifted a half word to the left.

For the sequenced hash code we would in practice use something like $k(c_n,k(c_{n-1}(,\ldots)))$ for a suitable integer function k(,). The current library uses k(a,b)=a+31*b, which may be fine for most purposes. One could argue that 31 should be substituted by some random integer, constant over the lifetime of any particular execution of an application.

These equality operations and hash codes have the apparent problem that the argument collections will usually be mutable objects and so the results of the operations are only valid momentarily, while conceptually they should be stable. We could require that the operations only be valid if the collection arguments were (flagged as) read-only (and throw an exception else), but that would make user code too inflexible and ugly. Thus, we will make it the user's responsibility to ensure that the collections are not mutated while being in a collection or while old equality results /hash codes are being kept. He is of course free to make collections read-only to guard against modifications.

The only natural comparer of collections is lexicographical comparison of sequences of comparable items. This is too special to include in this general-purpose library. (Set inclusion is not total).

5.2 Collection features

We will discuss collection features grouped according to whether they are expressible as interfaces or not. The features we touch on have been mentioned in chapter 2, or are present in one or more of the existing collection class libraries or similar libraries in some form.

The most fundamental feature of a collection arguably is the kind of items it can hold. As noted above, we will model that by letting the item type be the generic parameter of a collection class, or in case of a dictionary, the key and value types will be the two generic parameters to the class.

A feature that attaches to a complete collection must be classified by whether it is expressed by existence of certain operations, hence by interface implementation or by some other means. In the latter case, we will try to express the feature via a property (normally constant, but not necessarily); cf. how read only collections are treated below. One alternative would be to use empty marker interfaces like java's RandomAccess interface for a List class with a "fast" Contains method. Empty marker interfaces is in our opinion at best a misuse of the interface concept. Another alternative would be to use custom code attributes, but then the feature would necessarily be fixed by the implementation and access at runtime much less convenient than a property. Moreover, while code attributes would allow inspection at compile time, they would only be useful if development tools included specific support for them. A third, unsatisfactory alternative would be to just let the expression of the feature be part of the name or just be a documentation issue.

5.2.1 Interface definable collection features

The most important feature is the kind of data structure inside. We should not define an interface for every data structure we implement, but create a selection of interfaces by extracting and unifying sets of operations available in various data structures. We should not have a binary heap interface, rather a priority queue interface. The selection of interfaces and their interrelations is a global puzzle, where one tries to balance simplicity with being able to fit each data structure nicely into the structure. The result is presented below.

There is a major separating line in the set of collection interfaces between proper collection classes and dictionaries, because the latter by nature has function-like properties and not just set-like

properties. From an interface point of view, it is not so important that a dictionary can be considered as a special kind of set of pairs; that is more useful from an implementation point of view.

The extra interface definable collection features we have identified are: clonable, serializable, persistence, support for multiple collection operations and covariance of collections in the generic parameter.

5.2.1.1 Interleaved enumeration and updates: updateable views and fingers

It is illegal to interleave operations of an enumerator with updates to its collection. The linked list class of the java library has attached a ListIterator class that can iterate through the list, going forwards and backwards and safely perform some modifications to the list. We have tried to generalize this and combine it with the updateable view on a CLI ArrayList returned by the GetRange method or the result of a Java subList call.

The generalization consists of a few operations to create a view on a subinterval and move it around the collection. The view itself should be an updateable collection of the same type as the base type. For simplicity, a view is only valid as long as the base collection is only updated through that view. Since we only intend to implement this with list collections, we will just integrate the view functionality in the list interface.

An interesting and closely related concept would be a “finger” pointing to a specific place of a sequenced collection, an item or rather a gap between two items, and being kept valid under updates of the collection according to suitable rules. There would be operations of lookup and updates to the collection relative to a finger. The finger concept is well known in data structure literature, at least in connection with search trees. An updateable view is close to being equivalent to pairs of fingers.

5.2.1.2 Clonable

The CLI has a standard interface “IClonable” for marking this feature. It is explicitly not defined in CLI documentation whether a clone should be shallow or deep, but to support “IClonable” for collection classes, a general decision should be made on this point. The most natural would be that cloning logically corresponds to creating a new collection object of the same type and adding all the items of the original. The library should support cloning, but does not yet do so.

5.2.1.3 Serializable

Serializability in CLI is defined by applying the “SerializableAttribute” attribute to the class and the serialization process may be customized by implementing the “ISerializable” interface. This is the way to define serializability for our collection classes. Most of the data structures we use should support serializability “out of the box” by just adding the attribute, although the default serialization algorithms may not be the most efficient. We have not examined that question, but it might be faster to serialize an array of the items. The exception to straight default serializability is hash tables, which would have to be rehashed after deserialization because the hash codes of items are bound to have changed, at least if defined by the standard “object.GetHashCode()” method. The collection classes in the library ought to be serializable, but are not marked as such in this version.

5.2.1.4 Persistence

(This is the notion of persistence as used in data structure literature, cf. [DSST]).
If the base data structure of a collection class is persistent (fully or partial), it can be exposed as an interface in a couple of ways:

1. If the data structure maintains all versions, we could have a method or property returning old versions by a generation id and a method for returning the current generation id.
2. There could be a “Snapshot” method to remember versions on demand.

It depends on the type of persistence, full or partial, whether the old versions are mutable or not. In the case of full persistence, the “Snapshot” method can be regarded as a (fast) “Clone” method. While the first alternative is the typical viewpoint of the data structure litterature, the second one seems more natural in the realm of collection classes, so we use that (it can also be much more memory efficient). Since the property of being mutable is not an interface feature, the type of persistence should not interfere with the persistence interface.
Thus we could have an interface “IPersistent<T>” just with the operation “Snapshot” returning some sort of general collection type. But we would have to combine this interface with all kinds of collection interfaces resulting in a combinatorial explosion in the number of interfaces and we would have a too general return type of “Snapshot”. Instead, we create persistent versions of the interfaces corresponding to data structures where we actually implement a persistent version, in practice only leading to the “IPersistentSorted<T>” interface implementing “IIndexedSorted<T>” with a “SnapShot” operation returning an “ISorted<T>”.

The data structure literature defines other kinds of persistence, in particular confluent persistence, where one can combine several collection objects, e.g. in list catenation, in a persistent way. While interesting both from the point of view of the functionality offered to the library user and from the point of view of interface design, we will not attempt to implement such data structures and therefore we will not discuss the interface design.

5.2.1.5 Support for efficient multiple-collection operations

Such operations are abundant in data structure literature under such names as: catenate, merge or split and there are versions of heaps, lists, trees etc. with efficient operations of this kind. On of the virtues of linked lists as a data structure is exactly that it allows fast catenate and split operations. Unfortunately, many of the very fast algorithms are destructive, e.g. not preserving either of the two arguments in a catenate operation which makes it awkward to design as interface operations. Note that it typically is hard to combine such operations with persistence without a performance hit.

This kind of operations should be part of a library like this, but in the current version, we have decided not to include any implementations or design the interfaces. We have of course methods like “AddAll(items)” but they just enumerate and do not utilize any structure of the argument and are not “efficient multiple-collection operations” in the above sense.

5.2.1.6 Covariance in the generic parameter

The announced generic extensions for java 1.5 includes support for so-called “wildcards”, allowing one to partially consider C<U> as a subclass of C<T> if U is a subclass of T. This is interesting for a collection class library since it is natural to have an operation like

```
interface ISomeCollection<T> {
   …
   void AddAll<U>(IEnumerable<U> items) where U : T {}
   …
}
```

that will add all the U and hence T items of the argument to the collection of T. Unfortunately, the announced CLI generics will not allow a restriction on parameters as above although safe in this

case. We have decided not to introduce such methods into the interfaces without the parameter restrictions since the implementations would have to make runtime checks and casts.

5.2.2 Non-interface definable collection features

Here we discuss support for synchronization of operations, mutability, set vs. bag semantics, operation complexity, user-defined limited capacity, weak entries, stack or queue semantics of default add/remove operations and live file persistence. All these features have some counterpart in the java and/or CLI collection class libraries.

5.2.2.1 Support for operation synchronization

Any interesting data structure implemented as a class in CLI without multithreading in mind will be unsafe for simultaneous updates by different threads or by an update concurrent with a lookup operation. Some data structures, where the internal data are also modified by lookups (for optimization purposes), will even be unsafe for simultaneous lookups – the most notably example being splay trees. While the user is free to implement synchronization without support from the collection class library, the library can support it at least these ways:

1. Unconditionally protect internal data access with locks as in the original java library.
2. Like 1., but only lock if a certain flag property is set.
3. Offer wrapper classes that synchronize operations unconditionally with locks as in newer java libraries and early CLI.
4. Export an object for the user to lock on for synchronization. This is what newer CLI collection classes do.

The main problem with the first point is a large performance hit in the common situation, where the locking is not needed. The problem with the next two points is that this only supports a limited kind of locking scenario, where the locking is done at the granularity of a single (external) operation. Thus, a situation where one needs some sort of transaction synchronization over several operations is not supported. Moreover, the existence of this limited functionality may mislead non-experts into a false sense of security in an area that is notoriously difficult to test effectively.

While the last point is always an option for the user, the fourth point may be a relief in the case where the access to only one collection object has to be protected at a time by taking care of the management of locking objects. For this reason, we follow the strategy of newer CLI, with the same property name, “SyncRoot”. We have to be careful, that derived collection classes (like a read-only wrapper) exports the same object for locking as the base object.

Since it would be highly non-intuitive for many users to have to use locking to protect simultaneous lookup operations, we disqualify from this library any data structure that would need it.

Note that the discussion above shows that “synchronized” is not really a good feature for a collection object. We could introduce an “ISynchronizable” interface just containing the “SyncRoot” property definition, but since this is a universal property of at least editable collection classes, we will refrain from doing so.

5.2.2.2 Mutability: read-only or not, “fixed size”

A read-only collection compared to a similar type mutable collection supports fewer operations but is conceptually of the latter kind. Therefore, the concept is not definable by interface and we will define in collection classes with update operations a property, “ReadOnly”, which enables a safe

check at run time on the validity of update operations. When such a collection is read only, calling an update method on it will throw an “InvalidOperationException”.

There can be several reasons a collection is read-only. It may be a result of a query on another collection, where updating through the result set is not defined, but then the type of the result collection should not have update operations. It could be a collection, where the internal data structure is static, i.e. populated once and then only supporting efficient queries, not further updates. We do not have any such static data structures to implement in this work. Finally, the purpose could be to localize the code that can update the collection, to ensure some invariant or perhaps protect against untrusted code. (A thorough discussion of protection against untrusted code would involve a discussion of CLI code access security, which is outside the scope of this project.)

For the last of these reasons, a way is needed to make a read-only version of a collection. In implementation, this could be based on a shallow clone with “ReadOnly” set to true or could be based on read-only wrapper classes. With the latter, we will need one wrapper class per major collection interface, while the former would need code in every collection class implementation. We prefer the wrapper class method. The sought for functionality can then be exposed by an operation, say “ReadOnlyView()”, or by the wrapper classes having constructors taking the collection to protect as argument. We do not need to clutter the collection interfaces with more operations, so we select the latter, i.e. no special operations.

The CLI has a related concept, “Fixed Size”, of a dictionary object where the set of keys is immutable. While this may be interesting for similar purposes as having read-only views of collections, we have no obvious general-purpose applications of this in mind and have refrained from including the concept in the current version of the library. If included, it should be by means parallel to read-only.

5.2.2.3 Bag or set semantics

The distinction between bag and set is not well suited to be defined by an interface: conceptually, a set is also a bag, but bags could support more operations than sets, e.g. removing all copies of a certain item as opposed to removing just one copy of it.

Therefore, we define a property showing if a collection object is a set or a bag. We have chosen a Boolean “NoDuplicates”, since the meaning of that name should be immediately clear for the user.

5.2.2.4 Cost

Every collection class will have time and space performance characteristics depending on the behavior of the underlying data structure and the implementation details. These performance characteristics are mainly interesting for the programmer’s class selection and therefore the documentation, but some information may be useful at run time to optimize certain operations. The most obvious example is checking equality of the contents of two sets. We first check if the sizes are equal and if so, we check that every item in one set is in the other one. Here it is important that the set considered the other one has the fastest membership operation.

We add to every searchable collection interface a property, “ContainsSpeed”, taking one of the values constant, logarithmic or linear with the value based on the asymptotic complexity of the underlying data structure. One could construct other or finer grained cost properties, but we have only seen use for this one. Note Java’s marker interface, RandomAccess, with a similar purpose.

5.2.2.5 Stack or queue semantics of default Add and Remove

For a (random insert) list type, it is important what is the default position for adding and removing items. We have selected that Add will always add to the end while Remove will remove from the start or end according to the value of the Boolean property FIFO. This property will be modifiable by the user unless the list is read-only.

5.2.2.6 Fixed capacity/restricted size variants

That would be a collection class throwing an exception, blocking or dropping the item when an insertion is attempted after reaching a certain size threshold. While one could imagine less functional but slightly faster collection classes of this kind and special situations where this would be interesting, they do not belong in a library like this.

5.2.2.7 "Weak" collections

The main reason for considering this concept here is the standard java class "WeakHashMap". A proper collection class with weak references to the items doesn't make sense – what should be the result of enumerating the collection? Internal weak references could make sense for the entries of a dictionary as in the java class, but in our setting, dictionaries are always collections of entries. For this reason we will not implement "weak" directories in the library and refer users to the (easy) task of making a weak wrapper himself.

5.2.2.8 Live file persistence

(This is persistence as sometimes used in OOP literature.) This would be a feature to bind (or "tie" in Perl jargon) a collection object to some object on permanent storage so that the in-memory view is kept synchronized with the storage contents, across program invocations or application domains.

The concept is clearly interesting for programmers, but does not fit naturally into a general-purpose collection class library for CLI. Tying a collection object to an ordinary file system file does not suit the way CLI is normally conceived and functionality for tying to a DBMS table belongs to a data access library. In fact, ADO.NET contains functions to bind a database table to an object of type "DataTableMappingCollection" implementing the standard "IList" collection interface.

5.3 Interface architecture

There are two primary kinds of collection interfaces in the library:

- Interfaces capturing the essence of a family of data structures having a similar set of operations.
- Interfaces capturing the operations relevant for collection type results of queries on the first type.

To this we add intersections of interfaces of the first kind to clarify common functionality.

The interpretation of "family" and "similar" must balance the wish for few interfaces with the wish to not force too many "inefficient" methods onto implementations.

A main problem in the construction of the interface architecture is to avoid having to add unions of interfaces, typically unions of the primary interfaces with various small interfaces for auxiliary functionality. We do not need unions for implements clauses of class definitions. The need will appear if we have methods returning a collection that it is important to know implements two

(original) interfaces, not just one of them. Such unions, if there are several small auxiliary interfaces could lead to an explosion in the total number of interfaces.

The CLI (generic C#) puts a special role on the "System.Collections.Generics.IEnumerator<T>" interface in connection with iterator blocks; we must stay consistent with this special role. Else, we define all collection types independently of the "System.Collections" namespace.

All types are put in a single flat namespace, C5 (there are a couple of helper class violations of this rule now). We follow the C# language specification naming guidelines and use PascalCase with an "I" prefix to name interfaces. Names must of course reflect common usage and the most prominent features of that interface. Other identifiers then interface names will use PascalCase too except non-public fields, local variables and method parameters, which all use camelCase names.

5.3.1 Proper collections interface architecture

The following figure shows the system of interfaces for proper collection classes in the library. (Note: "IComparer<T>" is of course not a collection interface, but some of the collection interfaces implement it. For completeness, we should let "IEditableCollection<T>" implement "IHasher<T>".)

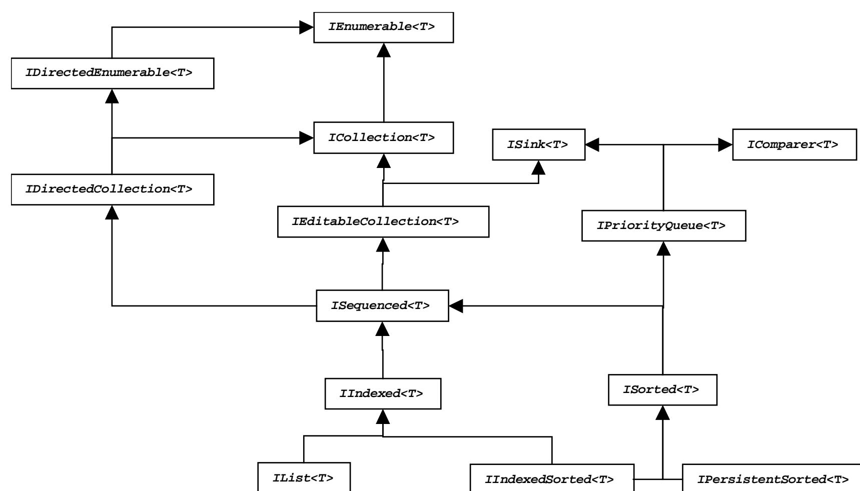


Figure 1: collection interface overview

We refer the reader to the appendix with the manual for the exact contents of the interfaces. Discussion of the contents is done for some part in this section and for the rest in the next section.

The primary collection interfaces are, with typical internal data structures:

- IEditableCollection<T>: hash tables
- IPriorityQueue<T>: heap data structures
- ISorted<T>, IIndexedSorted<T>: search trees and sorted dynamic arrays
- IList<T>: dynamic arrays as lists, linked lists.

5.3.1.1 Query result collections, enumeration

There are two basic interfaces for query results:

- IEnumerable<T> supports enumeration, GetEnumerator().
- ICollection<T> also "knows" its own size via the Count property.

Thus, one would return an IEnumerable<T> in cases where it would be expensive to decide the number of items in advance. IEnumerable<T> also has a few higher order methods (All, Exists, Apply) and ICollection<T> has CopyTo for copying the items to part of an array. These are almost the same as interfaces of the same name in the System.Collections.Generic namespace. These interfaces are also implemented indirectly by all "real" collection classes (except pure priority queues).

For query results on a sequenced collection, the result has a meaningful enumeration order and therefore a meaningful backwards enumeration order. In order to be able to do something like

```
foreach (T i in coll.Backwards()) {…}
```

to have coll enumerated in the opposite of the usual order, we introduce interfaces

- IDirectedEnumerable<T> and IDirectedCollection<T>

that adds a Backwards method as above and a Direction property showing if an object enumerates forwards or backwards relative to the original collection. That method and property are also present in ISequenced<T>.

Note: two enumerators of a collection of any type should deliver the "same" items in the same sequence if no updates come in between the creations of the enumerator.

5.3.1.2 IEditableCollection<T>, IPriorityQueue<T> and ISink<T>

IEditableCollection<T> is the simplest "real" collection interface in the library, i.e. describing a collection to which one can add, look for and remove items. IPriorityQueue<T> on the other hand is an atypical collection interface in that it does not even support (non destructive) enumeration or lookup (we do not want to add such operations to some of the target data structures). ISink<T> is just the intersection of the first two. The names IEditableCollection and ISink are terrible, but was the best we could think of.

5.3.1.3 IIndexed<T> vs. ISequenced<T>

These interfaces can be seen as intersection interfaces that both express primarily that a collection class is sequenced. The difference is that IIndexed<T> is used for collection classes that not only maintain a sequence order but also means of accessing items by index into the sequence (efficiently relative to other operations). Therefore, IIndexed<T> has:

- Properties for accessing items and intervals of items by index: this[i], this[start,end]
- Methods for finding the index of an item: IndexOf(item), int LastIndexOf(item)
- Methods for removing an item or interval: RemoveAt(int i), RemoveInterval(start, count)

ISequenced<T> only contains definitions for the sequenced collection equality and hash code methods. It is usually implemented by explicit interface implementation and not as ordinary virtual methods.

On the sorted collection side there is a parallel situation with the IIndexedSorted<T> vs. ISorted<T> interfaces. Here the indexed version in addition to the members above adds primarily

- Methods for counting items in ranges: CountFrom, CountFromTo, CountTo.
- New versions of RangeFrom, RangeFromTo and RangeFrom returning an IDirectedCollection<T> instead of an IDirectedEnumerable<T>
- Sorted versions of the higher order FindAll and Map operations.

(The last pair has nothing to do with indexing, but fits better here because the main sorted collections in the library actually are indexed).

It is moreover on the sorted side the data structure reason for making both sequenced and indexed interfaces lies: trees without sub tree sizes, like node copy persistent red-black tree snapshots do not allow efficient indexing and are therefore only sequenced. This also explains why the Snapshot method of IPersistentSorted<T> returns an ISorted<T>.

5.3.1.4 Why is there no stack or queue interfaces like in java and CLI?

One can use implementations of IList<T> for those purposes. Stack and Queue interfaces would not fit well into the architecture above.

5.3.2 Dictionary interface architecture

The dictionary interface architecture is much simpler:

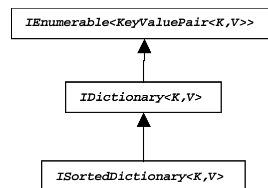


Figure 2: dictionary interface overview

We simply note that a dictionary consists of a set of pairs (with function property) and we have a plain dictionary interface for hash table based dictionaries and a sorted dictionary interface for search tree based ones.

5.4 Interface design (member selection)

We generally include in an interface all members, operations from the reference libraries belonging to the interface, perhaps after proper renaming. We have been through all java and CLI collection class manual pages to collect members. We even initiated a systematic cross-reference database, but it never got close to completion.

5.4.1 Some general naming and selection notes

We have tried to follow CLI naming but since we have decided to use “Range” only for ranges given by comparison of items, we use AddAll for AddRange, RemoveInterval for RemoveRange and GetRange that creates an updateable view on a list has become CreateView.

Some notable differences with java usage of words: we use Count where java uses size for the number of items in a collection. We use dictionary (like CLI) instead of java’s map. We use insert instead of add for insertions by position into lists.

5.4.2 Failure modes

While not formally definable in interface definitions of CLI, we must fix and document the correct failure modes for operations at the interface level to make implementing classes truly substitutable.

The failure mode can be specified as throwing an exception of a certain kind or returning a “normal” error code. One guiding principle for the choice is that out of range conditions in connection with indexing could be expected to be a result of a programming error so we throw an exception. Moving an enumerator after modification of the underlying collection must throw an exception. We differentiate between Add and Insert methods in that the former returns a status

indicating success while the latter will throw an exception on error. Finally, lookup and remove operations generally do not throw exception, but return status codes.

5.4.3 Higher order

The final version of the new CLI with generics will include some “higher order” functions, i.e. methods that take a delegate (sort of a function pointer) of a particular type and perform some operation that involves using the delegate on all items in the collection. There will be

- Exist and All that takes a Filter delegate and returns a Boolean,
- Apply that takes an Applier delegate and applies it to all items,
- FindAll that takes a Filter delegate and creates a new independent collection of the matching items and finally
- Map that takes a Mapper<T,V> delegate and creates an independent V collection of the mapped items.

We have chosen to define counterparts in this library, placing Exists, All and Apply on IEnumerable<T> and placing FindAll and Map on IList<T> and IIndexedSorted<T>.

5.4.4 Add vs. Insert

There in one major division in the range of operations that add one or more items to a collection: whether the item is to be put in a user-specified place in the collection (in that case, a list). We have decided that the user-specified-place operations will have an “Insert” prefix, all other adding operations an “Add” prefix or be just “Add” for a default addition operation. Now, what should happen if we try to add an item to a collection with set semantics and the item is already there? The adding operation needs to signal that situation, which can be done either via a return code (in the return value or in an out or ref parameter) or by throwing an exception. Both may be useful for the programmer, but since we do not want to promote exceptions for normal flow control, the return code method will be used for “Add” operations, returning a Boolean value. Note that for a collection with bag semantics “Add” will always return true. For “Insert” operations to a set, we have chosen to throw an exception when the item is already there. The idea being that the user is less likely to want to insert a specific item in a specific place in a set-semantic list and have to check a return code to see if the item was already somewhere else.

For operations adding several items to a set, an “Add” operation will just ignore items already in the set, while an “Insert” operation will throw an exception if any item is already in the set.

5.4.5 Lookup and combined operations

We have mentioned methods and properties for lookup by index above. The library supports the traditional membership operation:

```
bool Contains(T item)
```

This method tells if the specified item is in the collection; or more precisely if there is an item in the collection equal to the specified one. There is also a less traditional extended membership operation,

```
bool Find(ref T item)
```

This method works as Contains with the addition that if the lookup was successful, the actual item found inside the collection will be returned in the byref parameter. If T is a reference type, the item returned must be the exact garbage collected heap object found; if T is a value type, the item returned must be an exact binary copy of the struct found. This idea may seem a bit far-fetched at first glance, but is ideal for implementing a lookup inside a dictionary constructed as a set of pairs with equality of pairs defined by only the key parts. There is the problematic issue that a synthetic

collection of reference type items would have to cache the generated object to fulfill the specification.

We have included a number of operations that could be performed by combinations of Contains, Remove and Add, but has much more efficient implementations for many data structures by only having to traverse the data structure once:

- bool Update(T x): replace an item equivalent to x with x in the collection.
- bool FindOrAdd(ref T x): look for item and report actual object found, add if not found.
- bool UpdateOrAdd(T x): update item if found, add instead if not found.
- bool RemoveWithReturn(ref T x): remove x, report the actual object removed.

The existence of these operations make dictionary operations much more efficient. The two in the middle return true if x was added, which is probably counterintuitive with those names.

All the methods mentioned in this section belong to IEditableCollection<T>.

5.4.6 Multiple item operations

We have a usual collection of such operations: AddAll on ISink<T>; RemoveAll, RetainAll, and ContainsAll on; AddSorted for ISorted<T> and InsertAll for IList<T>.

5.4.7 Collection hashers

Slots for the unsequenced and sequenced collection hashers mentioned earlier in this chapter has been added as GetHashCode and Equals methods on IEditableCollection<T> and ISequenced<T>.

Classes implementing those interfaces must make sure that the right method variant will be called when a client calls it through an interface call. Thus, the methods should probably be implemented via explicit interface implementation.

5.4.8 Additional methods, properties, comments by interface

5.4.8.1 ISink<T>

This is the intersection of all collection interfaces having update methods. We have put Boolean NoDuplicates and IsEmpty properties and the SyncRoot property here. There is a Check() method for a consistency or invariant check of the internal data of the collection class.

5.4.8.2 IPriorityQueue<T>

The priority queues this interface describes are double ended by having methods for reporting and deleting both the maximum and minimum: FindMin(), DeleteMin(), FindMax(), DeleteMax(). Note that we use the traditional priority queue names ([CLRS]) – the usual conventions of this library would prescribe using Remove instead of Delete. Originally, we also envisioned a single ended priority queue, but could not choose which end to support. The user should not be forced to code around only having, say, a queue with only maximum-based operations. To be able to implement a traditional binary heap we have added a Boolean DoubleEnded property to the priority queue. Crippled implementations will signal that it only supports maximum-based operations by setting the property to false and throw exceptions on invocation of the minimum based methods. This violates our design principles, and we intend to remove DoubleEnded and the binary heap implementation.

Note that [CLRS] also has an IncreaseKey operation in their definition of an abstract priority queue. We have found it hard to support such a method in this setting.

5.4.8.3 IEditableCollection<T>
In addition to members mentioned above, IEditableCollection<T> contains the properties IsReadOnly and ContainsSpeed, the standard ToArray method and a couple of methods interesting only for bags: RemoveAllCopies, CountItems.

5.4.8.4 ISorted<T>
The remaining methods of the sorted interface are four strong and weak predecessor and successor operations and a special "Cut" method. The predecessor and successor methods throw exceptions to indicate no such item. They were originally put in an IPredecessorStructure by themselves, but we found no real utility of such a separate interface.

The Cut method takes a non-decreasing function on the item type, given as some object (formally) implementing IComparable<T> and reports the largest item at which the function is negative and the least item at which it is positive, if any. If T is actually IComparable<T>, the function object, t, is a T item and the collection uses the natural sorting order, Cut simply performs a search for t, reporting predecessor and successor if any – without throwing exceptions if none were found.

5.4.8.5 IList<T>
This interface is for sequenced collections where the user decides the sequence order.

In addition to already mentioned members, there are a selection of Insert methods: by index, at either end and before or after another item, which would have to be found first.

Second, we have methods for the view functionality: CreateView for creating a view, Slide for moving the view relative to the base list and the properties Base and Offset for reporting the base list and offset of a view.

Finally, there are the following special methods for changing the sequence order:
- Reverse for reversing the sequence order of the items.
- Shuffle for randomly shuffling the items.
- Sort for sorting, IsSorted for testing if sorted – according to some comparer.

5.4.8.6 IDictionary<K,V> and ISortedDictionary<K,V>
These have an indexer and the expected methods for maintaining the dictionary and a few non-standard combined ones, like bool UpdateOrAdd(key,val), which does the same as "this[key]=val", but also report whether the key was already in the dictionary.

Note, that unlike java, we do not allow searching for values in a dictionary, only for keys.

The ISortedDictionary<K,V> just adds predecessor and successor operations.

6 Class design and algorithmic design

In the previous chapter, we described the architecture and details of the interfaces of our library. Here we will describe the actual selection of data structures implemented and the classes that do. First, we discuss several topics important for all several collections classes, then we cover the implemented collection classes and finally discuss the support classes of the library.

6.1 General implementation topics

We partition these topics in topics of classes and methods, semantics, complexity and security.

6.1.1 Classes and methods

6.1.1.1 Helper classes

There will be the need for other classes than the collection classes proper. Helper classes needed for the results of query type operations should be nested in the original class. We will not implement factory classes, say a factory class that could create collections with a certain interface and where the specific implementation would be given by name. While this could enable very easy substitutions of implementation in user code, it would mask differences exposed by the constructor sets. The user may of course implement own factory classes easily and in such a concrete setting, the use of specific parameters to constructors should be more obvious.

6.1.1.2 Static methods

We avoid static methods (utilities) when the same effect can be achieved with an ordinary instance method. Therefore, an operation like sorting a list would use a “Sort()” operation of the list interface and not (as in Java) a “static Collections.Sort(IList lst)” method.

6.1.1.3 Natural and/or external item operations (hashers, comparers)

For all item types, there is a natural notion of item equality and hash value – given by the “Equals” and “GetHashCode” methods inherited from the “object” class or given by methods defined as overriding those. For item types that are internally comparable, by implementing the “CompareTo” method of the standard or generic version of “IComparable”, there also is a natural notion of item comparer. In these cases, we can implement “natural” collection classes containing (virtual) calls to the mentioned methods. However, sometimes we need collection classes where the hasher or comparer to use is not the natural one or there is no natural comparer. The hasher and/or comparer must be supplied to the collection class constructor. Unfortunately, we cannot use the exact same code as the above mentioned “natural” collection classes, the calls to the Equals etc. methods has to be changed to interface calls to methods of the “IHasher” or “IComparer” generic interfaces.

We have chosen to only implement collection classes with general hashers and comparers. The classes will have constructors, where one can supply a hasher or comparer as relevant, but also constructors with no such arguments. Those constructors will construct a hasher or comparer from the natural notions if possible – if we need a comparer, but none is supplied and the item type is not “IComparable” then the constructor will throw an exception. The workings of the hasher and comparer construction will be described below in the support classes section.

This choice has the disadvantage that we loose some performance at the item operation calls for natural collection classes, the worst case probably being that in an integer collection, integer comparisons etc. will not be inlined, but called through an interface.

6.1.1.4 Collection hashers
The unsequenced and sequenced hash codes of collections have been defined in chapter 5. They will be returned by the interface specific method definitions mentioned there. There could be three different ways to implement them:
- Compute on demand
- Compute on demand, cache result
- Compute incrementally while modifying the collection

The first one is clearly too expensive, violating gravely the contract that hashers should be of O(1) asymptotic complexity. The third one, while securing the correct complexity, will spend time maintaining hash codes that will only be used if the collection happened to be inserted in some other collection, an uncommon case. If a collection to be inserted in another one is not changed after the first time the hasher is called, then the second way in the list will only use constant time for computing the hash code, when amortized over update operations to the collection. We will choose the second way of computing the hash codes, with the additional comment that we protect the cached values with stamp ids and recompute the hash codes if the collection was modified since the last computation of that hash code. Note that we do not (cannot) check whether the individual items of the collection (on whose hash codes the collection hash value depends) were changed.

Note: A collection may contain itself, perhaps indirectly, if the item type is object. In this case, the collection hashers would never return but get into an unbounded recursion and abort by throwing an exception (probably StackOverflow). From a strict point of view, this means that our implementation is in violation of contract, but protection would incur a gigantic overhead to maintain a (hash) table of already encountered objects to detect loops.

6.1.1.5 Constructors
The principles for selecting which constructors a collection class will have are the following.

First the standard parameters that can be present. There can be constructor parameters of external hasher and if relevant, external comparer. If the collection class uses arrays internally for storage, there can be a parameters asking for an extended start capacity. There can be non-standard parameters for special data structure parameters. There will be no parameters for populating the collection class.

As for constructors, there will be a default one and one for each combination of the standard parameters. With non-standard parameters, the precise selection of constructors will be made case by case.

6.1.1.6 Use of protected fields
Sharing code among collection classes, we have found it necessary to also have a number of protected fields. For example, the size and stamp integer fields for all the collection classes in the library are defined in the generic “EditableCollectionBase” helper class. That way, subclasses can have fast access to those values. Unfortunately, this means that user-created extensions of the collection classes will have uncontrolled access to the fields and may by programming error violate invariants as “the size field holds the number of items in the collection”. This can be considered bad design and is at least questionable in view of the requirements of chapter 3.

6.1.2 Semantics

6.1.2.1 Relations between set and dictionary implementations
Conceptually, a dictionary is a special kind of set of pairs, namely one defining a (finite, partial) function from the key domain to the value domain. Now, if we use the equality concept for pairs

that two pairs are collection equal if their first components are equal, then sets of pairs will correspond exactly to finite partial functions i.e. dictionaries. We shall in fact implement dictionaries by building on set implementations in this way. Details are given below in the treatment of class "DictionaryBase".

An opposite way of relating set and dictionary implementations would be to use a dictionary with a value domain of e.g. integer and all actual values equal to zero inside a set. This has the advantage of not using the slightly non-standard pair equality, but the disadvantage of wasting space for unused value components in the set class. Note on the other hand that we may use this construction with counts as dictionary values to build a bag collection over a dictionary.

6.1.2.2 Keeping objects alive

The question is: is it acceptable for a collection class to be implemented in a way that objects are kept alive longer than their "logical" lifetime? This could happen by design, using some lazy deletion strategy, where items to be removed are not necessarily deleted from the internal structure at once, only marked as removed, postponing the actual removal for later. It could also happen by error, if internal data structures (e.g. arrays) are not cleaned between operations. Special care must be taken with unused elements at the end of the internal array of dynamic array data structures.

The main principle is that object lifetime should be predictable by the library user and not counter-intuitive. Thus, the normal policy is that collection classes do not keep objects alive that are not reachable via external operations.

We will allow implementations keeping objects alive past their formal application lifetime if this is in concordance with the assumed normal usage mode. The deviation from normal policy must be clearly stated in documentation together with a description of the assumed usage mode. There is in fact only one case of this in this library, persistent red-black trees.

6.1.2.3 Set vs. bag classes

In an ideal world, we would be able to specify whether a collection class had set or bag semantics independent of any other features. In practice, some data structures (e.g. pure linked lists) have much more efficient bag than set implementations and we would not want to make a set version. One could imagine data structures, where we could have a set and a bag implementation in the same class, selectable by a flag property, but we have not recognized such cases and all classes will be either sets or bags in this library.

6.1.2.4 The meaning of the "Update" method

The meaning of the "Update" method may be mysterious from a purely conceptual point of view. Assume we have a set collection and the corresponding collection equality (not necessarily reference or binary equality). If item c is in the set and d is collection equal to c, then the result of "Update(d)" will be that the actual item equal to c inside the collection object is replaced by d. More precisely, if the item type is a reference type, the reference to c inside the collection object will be replaced by a reference to d; and if the item type is a value type, the copy of c inside the collection object will be replaced by a binary copy of d. For bag collections, the question is more difficult and will be discussed in the next subsection.

6.1.2.5 Bag detailed semantics

The semantics of a bag collection under a collection equality that is not reference equality (for objects) or exact binary equality (for value types) needs some consideration. Assume the items a, b and c are equal under the collection equality and that we add a and b and then remove c. Then afterwards, querying for either of the three should result in the answer that the collection contains one copy of that item, but which of the items should actually be kept inside the collection object?

The best we can do is to state that the answer is undefined – "implementation dependent" might be too strong since the answer would depend on the precise order of operations.

Thus, when collection equal, but not binary equal, items have been used in update operations of a bag collection, it will be uncertain exactly which of them should be in the collection object, only the number of equivalent items. We shall stretch this uncertainty so far that we accept bag collection implementations that just keep a single representative item together with a count for each equivalence class. Note also that we will allow the "Update" method to either update a single copy or all copies at the discretion of the bag collection class implementer.

6.1.3 Complexity and tradeoffs

6.1.3.1 Virtual methods or not

Should we make interface-defined methods virtual in collection classes? If virtual, inlining is not possible and there will be a performance hit on calls to the method. Microsoft seems to find that important, although it probably is only discernible for a collection that is a thin layer over an array. However, if we do not make the methods virtual, we would have to make them or the collection classes sealed, else subclassing will be too error-prone for the user. Therefore, we make all interface-defined methods virtual.

6.1.3.2 Code duplication vs. code sharing

In general, it should be considered more acceptable to duplicate code in a library (to avoid method call overhead) than in normal application code since it should be possible in this setting to cope with the maintenance question because changes are rare. On the other hand, changes will only be rare when the implementation has matured and so one should share more code in the beginning and ideally only duplicate according to actual measurements of how much performance can be gained.

6.1.3.3 Recursion vs. iteration

Some of the data structures may have an original formulation in recursive terms, e.g. search trees, heaps and sorting algorithms. It is often possible to give an iterative formulation without it becoming overly complex even if having to maintain an explicit stack. From the outset of this project, we have little experience to judge the balance between implementation complexity and less method call overhead – experiments had to tell. We have to be on the guard for recursive algorithms, which might lead to stack overflow under unhealthy circumstances, like pathological data for a fully recursive quicksort.

6.1.3.4 Implementation trade-offs (e.g. space/time)

Can we give general rules for how to balance tradeoffs in a collection class? Imagine a class, where we can choose to do some consistency check on every operation or to do a more extended check but only on update operations (this is relevant for list views). Another case is using more space to speed up lookups (with the same basic data structure). In principle, we cannot make a general rule here, because the best trade-off depends on the final use of the class, which cannot be known.

One could argue that if a class has very fast lookups by index (a dynamic array), then the user would have chosen it for exactly that reason and we should avoid compromising lookup. We might as well argue that exactly because of the fast basic lookup speed we can afford to spend a little there to save on space use or improve update speed.

6.1.3.5 Expense on external operations

The complexity of collection class operations cannot in general be stated precisely based only on library code, because there may be calls to external objects, hashers or comparers, involved. In principle, one should state complexity in terms of internal cost plus number of external operations. In practice, we just state the total complexity assuming external operations have O(1) cost.

6.1.3.6 Shrinking of internal arrays

When we use arrays internally in collection classes to hold items (directly or indirectly), we have to expand them from time to time by copying to larger ones if we keep adding to the collection. The question is: should we shrink the arrays (by yet more copying) if we at a later stage remove many entries? In practice, this will be relevant here for dynamic arrays and hash tables.

First note, that if we do not let the arrays shrink we would loose the linear space use if interpreted in the sense that space use is linear in the current number of items and not just that space use is linear in the maximum number of items since the creation of the collection. It is not entirely clear which sense is the most intuitive – the conservative choice would be the former.

Secondly, if we do shrink, we shall have to build in some laziness into it. If we double the size on each expansion and only add items, the array would never be more than double the necessary size, but if we try to keep this bound for shrinking, we could end up copying the array at almost every update if they happened to take place right at a threshold. Thus, we are bound to have much larger than a factor 2 space overhead if we do shrink.

We choose *not* to shrink for the following (additional) reason. We do not expect the user to use a collection class in such a way that it is first filled with many items, then almost emptied and kept around with few items for an extended period. Either it will be emptied and/or discarded soon or it will soon grow to a large size again. In these scenarios, shrinking the arrays will be a waste of time. In the expensive, exceptional case, the user should "manually" shrink by copying the lasting collection to a new one (and a warning put in the manuals).

In any case, unused array entries should be overwritten with null or T.default to avoid retaining objects that are otherwise unreachable

6.1.3.7 Space use

We require that space use of a collection class should be linear in the number of update operations performed since the creation of the collection object. Here multiple item operations should be counted with multiplicity.

Many classes will have a stronger space use property, namely that the space use is linear in the actual number of items. The exceptions are classes that use arrays for internal storage and red-black trees when used persistently. The former case respects the space bound of being linear in the maximal historical size, while the best known (to us) space bound of the persistent red-black trees in terms of the actual number of accessible items is superlinear.

6.1.3.8 Interpolation of data structures

By this, we mean using two (or more) data structures to implement a collection class, using one for small item counts, and another one for large item counts. This should only be perceived after benchmarking collection classes built on the two data structures and looking for a pair, where one could hope for a good fit. For the same reasons that we will not shrink internal arrays, we should not transform back to small-size data structure on deletions. In practice, we have not seen any real opportunities for such interpolation.

6.1.4 Security and safety

6.1.4.1 Fault tolerance

While the exact state of a collection class after an exception was thrown by an operation may be unspecified, the internal structure should always be consistent. The highest risk in this case would

be optimized bulk update operations, where one might encounter illegal input in a state where it would be expensive to reestablish a consistent partially updated internal state before throwing the exception. Since the user would be unlikely to want to rely on the collection anymore, an action could be to clear the collection and set an "inconsistent" flag and let the collection throw exceptions on any subsequent operation. The bulk update operations in the current version of the library need to be checked with regards to this question.

6.1.4.2 Thread safety of simultaneous queries

Many users would not even consider the possibility that simultaneous queries to a collection class would need to be locked to be thread safe. Since we require that the functionality of the library must be "intuitive", simultaneous queries must always be thread safe. While one could argue that one could just write a warning in the documentation, it would be hard to do in a way so that non-experts would recognize it with certainty. This disallows most notably splay trees as base data structure.

6.1.4.3 Stamps to detect modification

There is a variety of scenarios, where a piece of code has to check if the collection has been changed since some other occurrence. If so, the reaction could be to recompute something or throw an exception. An enumerator has to check if the collection was changed since the time the enumerator was initialized. Collection hash code caches must check if the collection was changed since the cached values were computed. A view into an "IList<T>" has to check for direct updates to the base list. A range or interval class must check for updates to the base collection. In all these cases, we increment a protected integer field, named "stamp", in the collection class on every update and save copies of "stamp" for comparison wherever an update check is needed. This seems to be the cheapest way to achieve the effect. With the current 32-bit implementation of CLR, there is a possibility for long running code to have a false negative result of the check, although the probability would be quite small for anything but a carefully crafted test case. We could eliminate the risk with a 64-bit long field but avoid it for the (assumed) performance hit in the current CLR.

6.1.4.4 Read-only versions/views of collections

All collection classes will have a "ReadOnly" property based on the "isReadOnly" protected flag field of "EditableCollectionBase". Every update operation checks the value of the flag and throws an "InvalidOperationException" on error. While this could be used together with shallow cloning to implement read-only versions, it is only used for protecting red-black tree set snapshots from changes. All other read-only versions are implemented via wrapper classes.

6.1.4.5 CLI code access security

The library has not been marked with the "AllowPartiallyTrustedCallersAttribute" and therefore cannot be used from partially trusted code (e.g. code executing from local intranet or internet zones). On first glance it seems unlikely that this library should contain code that would allow partially trusted code to circumvent security, but the library has not been thoroughly audited making sure it follows the .Net secure coding guidelines.

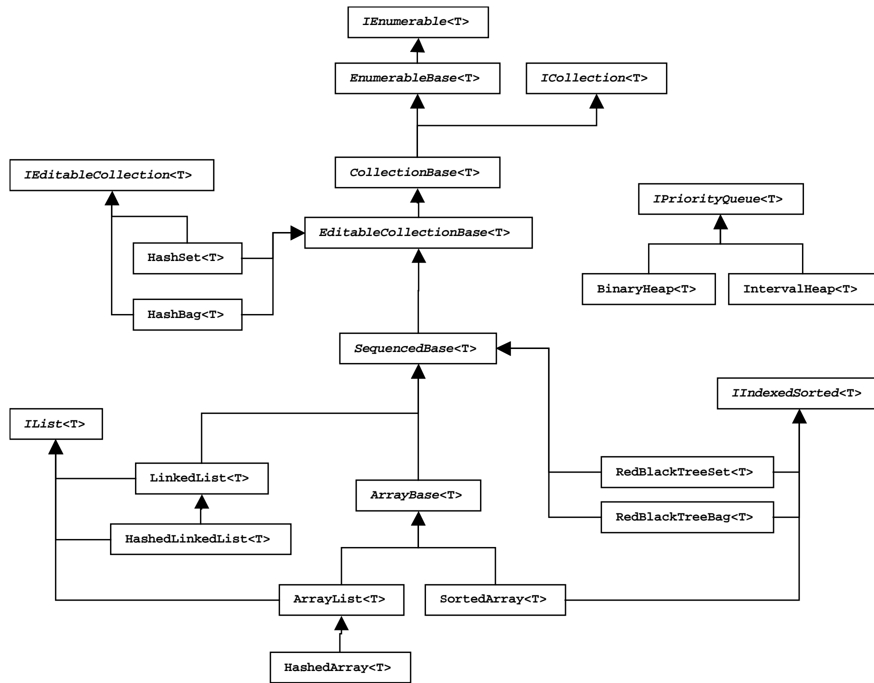
6.1.4.6 CLS compliance marking

We do not apply the CLSCompliantAttribute to the library assembly, and so all types will implicitly be marked as not CLS compliant. This is correct since all (interesting) types are generic and therefore not CLS compliant.

6.2 Collection classes

In this chapter, we describe the collection classes of the library with specific details on their implementation and why the particular data structures inside were selected.

The following diagram shows the inheritance and implementation hierarchy of the collection classes. Note that we only show “direct” interface implementation by classes, the interface hierarchy is shown in chapter 5. Neither do we show (nested) helper classes.

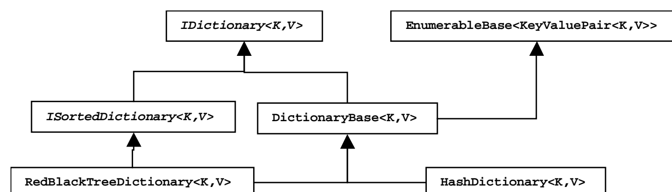


The figure shows a string of base helper classes implementing some common functionality: from EnumerableBase to ArrayBase.

We will describe this structured by data structure type and then by helper class kind. From the outset of this project, it was the intention to implement several different data structures of the various kinds and compare their performance. The only place we have done so is for heaps. For the other data structure kinds we have only managed to implement one, the most prominent one in the literature, though in many cases trying a number of minor variations.

We use three algorithms the reader is not assumed familiar with: a list order labeling algorithm, node copy persistence for red-black trees and interval heaps. These will be explained together with the class implementing them.

The following figure shows the similar picture of dictionary classes:



There are only two dictionary classes: based on the HashSet respectively RedBlackTreeSet classes by the procedure described in a section earlier in this chapter.

6.2.1 Hash tables

The library code implements several variations of linear hashing, namely linear probing and linear chaining [CLRS] in combinations with reference type bucket cells and value type bucket cells. The compiled code only contains one variation determined by preprocessor flags. See the description of class HashSet for more details. The choice of trying these variations of linear hashing was based on the findings in [PR]. The implementations have the usual theoretical complexities O(1) expected for lookups and O(1) expected amortized for update operations. There are other linear hashing strategies like double hashing, but we have not implemented those.

Based on the HashSet implementation there is also implemented a bag collection, HashBag, and a dictionary, HashDictionary.

We would have liked to include an implementation of perfect hashing with worst case O(1) lookups, e.g. the in practice very efficient “cuckoo hashing” ([PR]), but there are some points to consider. For perfect hashing, we need an item hasher that is guaranteed to separate items that are not collection equal (while being consistent). For a reference type, not overriding “GetHashCode()” or “Equals” from object, the runtime will create unique int hash codes. For other types the domain may very well be too large to have a fixed injective function into 32-bit ints. So we would need an “ExtGetHashCode()” function mapping the domain injectively to, say, an int array and let the perfect hashing datastructure work on such hash codes. We planned to create a tool for creating such extended hash functions for value types (with binary equality). This could be achieved by using reflection, but the performance hit would preclude its use. Better would be to create the function via the CLI runtime code generation (RTCG) library, but in the available version of the SDK, RTCG did not work well with generics. Note that perfect hashing data structures does not handle clashes of hash codes gracefully.

6.2.1.1 HashSet class

The main internal data of the class is an array with elements of nested class “Bucket”:

```
    Bucket[] table;
#if !REFBUCKET
    bool defaultvalid = false;
    T defaultitem;
#endif
#if REFBUCKET && LINEARPROBING
    class Bucket
    {
        T item;
        int hashval; //cache!
    }
#elif REFBUCKET && !LINEARPROBING
    class Bucket
    {
        T item;
        int hashval; //cache!
        Bucket overflow;
    }
#elif !REFBUCKET && LINEARPROBING
    struct Bucket
    {
        T item;
        int hashval; //cache!
    }
```

```
#else  //!REFBUCKET && !LINEARPROBING
    struct Bucket
    {
        T item;
        int hashval; //cache!
        OverflowBucket overflow;
    }
    class OverflowBucket
    {
        T item;
        int hashval; //cache!
        OverflowBucket overflow;
    }
#endif
```

We have shown the definition in terms of two preprocessor symbols, LINEARPROBING and REFBUCKET, which determines respectively if the code uses linear probing or linear chaining and if the (first) bucket is of reference type or value type (the actual code is less readably organized). With linear chaining, the buckets belonging to a specific position in the table are chained together via the "overflow" fields. The hash codes (results of "GetHashCode()") for items are cached in the hashval field. The (trivial) constructors have been elided for clarity. The fields are internal and not private because the surrounding HashSet class needs to access them directly. If the first bucket is of value type, we treat values equal to T.default specially using the defaultitem and defaultvalid fields.

From the outset, one would guess that linear probing and value type buckets would put the least pressure on memory management, but use code that is more complex. Value type buckets waste most space in empty table elements. Indeed, performance measurements (chapter 7) indicates that linear probing and value type buckets is a little slower at small table sizes, but behaves better at large table sizes, and therefore the default compilation uses that combination.

The table size always is a power of 2 and the index for an item (bucket) is computed from the hash value using the universal hash function family mentioned in [PR]:

index = (k * hashval) mod tablesize

Here "k" is constant over at least the lifetime of a specific table array. There are preprocessor flags (RANDOMINTERHASHER and INTERHASHER) that determine whether k is 1 (a bad choice), a large odd compile time constant (better) or a random odd integer. The last choice will imply the good, randomized expected complexity asymptotics.

The main non-trivial work in the code is done by the private methods hv2i, resize, searchoradd, remove and clear. The hv2i method computes indices as described above; resize expands the table and rehashes the buckets. The searchoradd method follows the structure searching for the place of an item and reports it, add it or replace it as needed for the Contains, Find, Add etc operations. Likewise, remove does the core work of the Remove and RemoveWithReturn methods. Remove for linear probing uses the algrithm "R" of [Knuth3] as suggested in [PR].

HashSet has one more nested (enum) class, Feature, which is merely a debugging aid.

6.2.1.2 HashBag class

The HashBag class is a straightforward implementation on top of HashSet of a bag collection as a wrapper of a set of (item,count) pairs.

6.2.1.3 HashDictionary class

The HashDictionary is a straight application of the DictionaryBase class to create a dictionary based on the HashSet set collection class.

6.2.2 Dynamic arrays

The library contains a straightforward implementation of a dynamic array as a list collection and two variants: one that keeps the items in sorted order and one that uses an auxiliary hash dictionary as an item to index mapping to speed up lookups.

The plain dynamic array, "ArrayList", by nature has bag semantics, while the sorted and hash indexed variants by nature have set semantics.

We have earlier discussed the reasons for not shrinking the internal array. There is an update time vs. space tradeoff possible with expansion, but for no specific reason, we have chosen to simply follow tradition and double the internal array when filled.

The most interesting variant not present is a persistent variant of the plain dynamic array, trying one or more of the designs of data structure literature ([DSST]). At first glance, a variant with a (sorted) tree index instead of a hash index would have no benefits compared to the sorted array.

6.2.2.1 ArrayList class

The array list data structure combines constant time lookups by index with linear time lookups by value and linear time general updates but constant (amortized) time for inserting and removing from the end.

The ArrayList<T> class is a list collection based on a dynamic array data structure. The implementation is based on the ArrayBase<T> helper class. Most of the specific functionality in the implementation is collected in the protected methods "expand", "updatecheck", "modifycheck", "insert", "addtosize", "indexof", "lastindexof" and "removeAt". Of these, the first four overrides methods of the ArrayBase<T> helper class. The updatecheck method is called from all public update methods to check if it is valid to access the list for update, while the modifycheck method is called in all other public methods to check view validity (see later). The method addtosize has to do with keeping size information synchronized in views.

Modification operations moves end segments of the array via the System.Array.Copy method assumed to be more efficient for long segments than a for loop. This may be less efficient for small lists than not using System.Array.Copy, but we have not made measurements to compare.

For the first library implementation of the updateable view functionality, we only have limited updates: a view becomes invalid if the base list is modified by other means than through exactly that view. Moreover, when we create a view on top of a view, it actually becomes a view directly on the original list. The most interesting aspect of the implementation of view functionality is keeping track of the validity of views using modifycheck and updatecheck methods.

Most other public method implementations are straightforward. We will just mention Sort that employs our implementation of IntroSort and Shuffle that uses our C5Random random number generator.

Because we implement HashedArrayList<T> as a subclass of ArrayList<T>, we have made a couple of fields protected: baselist, basesize and fIFO. This has the same engineering issues as the protected fields of the base helper classes.

In the current implementation, a couple of methods have suboptimal running time asymptotics, namely RemoveAll, RetainAll and ContainsAll. Their running time would be proportional to the product of the size of this array list and the number of items in the collection argument. An implementation with an auxiliary hash table should enable a running time linear in the total number of items, though with worse constant factor in space use.

6.2.2.2 SortedArray<T> class

The sorted array data structure combines constant time lookups by index with logarithmic time lookups by value with linear time updates.

The data and basic dynamic array infrastructure methods are defined in the base class ArrayBase<T>. Most operations are simple dispatchers of calls the private methods binarysearch and indexof and the system utilities Array.Copy and Array.Clear. The implementation of "Cut" is a little more complicated, though straightforward.

For some of the "XxxAll" methods there are issues with regards to the asymptotic complexity. The RemoveAll and RetainAll are O(m*log n), where n is the size of the collection and m is the size of the argument. There are alternative implementations that are e.g. O(m+n) and could be advantageous for some ranges of (m,n) values. The amount of extra space needed also varies. We have not examined these questions as thorough as one could wish.

The AddSorted (and AddAll if we sort the argument first) have similar issues depending on the sizes of n and m. We do a straight array merge, but [Knuth] discuss merge methods handling situations where n and m are of different orders of size better.

6.2.2.3 HashedArrayList<T> class

The hashed array list data structure combines constant time lookups by index, expected linear time lookups by value and linear time general updates but constant (amortized) time for inserting and removing from the end. It has formally the same or better asymptotics than the plain array list, but worse constant factors because of the hash index updates.

HashedArrayList<T> is implemented as a subclass of ArrayList<T> adding an index field of type HashSet<KeyValuePair<T,int>>, i.e. (the inside of) a hash dictionary. Many methods, Xxxx, are simply implemented as a call to base.Xxxx followed by a reindexing of part of the hash index. Apart from that, the implementation is built around a few protected utilities: the new reindex(start, end) and the overrides indexof, lastindexof, insert and removeAt.

Alternative implementations: a class with an array list field and a hash dictionary field but that seemed to require much access into the internals of those classes. An extended duplicate of the array list code should enable better performance, but was not considered for this first version.

6.2.3 Linked lists

The library implementation of linked lists is based on a traditional doubly linked list data structure with a couple of less standard additions.

The plain linked list has natural bag semantics, but we have implemented a variant with set semantics based on a hash index. Making a hash index and the view functionality of the IList<T> interface play together on linked lists was a bit challenging as will be described below.

A list implementation based on a singly linked structure would save space in the nodes, make some operations faster and other ones much slower. There may be applications where only the faster operations are needed, but we think the inclusion of a singly linked list implementation would make the library less user friendly – either having the same interface and quite different performance properties or having a reduced interface cluttering the type structure.

It would be interesting to try a blocked linked list variant, which might help performance by lessening the pressure on memory management. Another interesting variant would be persistent linked lists – we only do persistent trees, see later.

6.2.3.1 LinkedList<T> class

The complexity of the linked list is characterized by linear time lookups by item or index, linear time updates with the remark that operations at the ends of the list will be performed in constant time.

There are three nested classes, Node, Range and TagGroup. The Range class contains straightforward enumerator and interval query functionality. The TagGroup class will be discussed in the next section. The Node class is the basic building block of the list:

```
class Node
{
     Node prev, next;
     T item;
     //For tagging
     int tag;
     TagGroup taggroup;
     …
}
```

The tag and taggroup fields should be ignored for now, which leaves a standard doubly linked list cell. The data of the list class are:

```
class LinkedList<T>
{
     Node startsentinel;
     Node endsentinel;
     //For view:
     int offset;
     LinkedList<T> baselist;
     //For tagging:
     bool maintaintags = false;
     int taggroups;
     …
}
```

The list class has two synthetic sentinel Nodes that act as predecessor of the first node respectively successor of the last node in the list and are the links from the list class to the actual data structure. The first implementation used “first” and “last” references from the list class and null instead of

sentinels as prev/next fields at the ends, but the sentinel idea simplified the code considerably after it had been complicated by introducing view functionality.

Most single item public methods are simple using the following utilities:

- updatecheck, modifycheck: used like the dynamic array ones
- insert(succ,item), insertNode(succ,newnode), remove(node)
- find, dnif, get(n): return node by item search (either way) or index.

However, most multiple item methods are terribly complicated, though not difficult. Some are also suboptimal or of questionable complexity (the same as the dynamic array implementation).

The Sort method is implemented by a port from java of a stable, in place merge sort by Peter Sestoft.

The Reverse method is implemented by swapping items and not nodes, since that involves much less updates, and only should be less efficient for items of large value type.

6.2.3.2 A list order labeling scheme

Using a hash index from item to node with a linked list poses a problem in connection with the view functionality that we did not see in the similar situation of dynamic arrays. The problem is that when we lookup an item in the index, how do we see if the node is inside the view without an expensive search through the list? The solution we have chosen is to use a list-ordering algorithm of Sleator and Dietz [SD] in the version described by Bender et al [BCDFZ].

The basic idea is simple. We give each node an integer tag and make sure the tags are always in increasing order. Firstly, whenever we insert a node in the list we give it a tag that is between its neighbors, say the mean of the two. If there is not enough room, we redistribute the tags of part of the list to make room. The only problem is how to choose the amount of renumbering to balance with the frequency of renumbering. The answer of [new list order] is the following. When we run out of room we follow the list both ways comparing binary prefixes of tags, counting the segments of the list that have common prefixes of length w-1, w-2, …, w-b,… where w is the word size (32). Let k be a parameter to be defined. For the least b where the segment is of size at most k^b, we redistribute the tags in that segment evenly. We need $n<=k^w$ to be sure such a b is found and k<2 to be sure there are sizable gaps after tag redistribution, so we choose $k=n^{1/w}<2$.

The cost of this redistribution is $O(k^b)$. Note that before this redistribution the segment of common prefix length w-(b-1) had at least k^{b-1} nodes, now it has $k^b/2$. Thus there must be at least $k^{b-1} - k^b/2 = k^b (2-k)/(2k)$ insertions before we redistribute this segment again, and the amortized cost of redistributions of tags at a given b is O(k/(2-k)) per insertion into the list. As long as n is not close to 2^w, this is O(1) per distinct value of b and hence the total amortized cost of redistributions of tags will be O(w) pr insertion into the list. Note that the typical case of always inserting at the end of the list seems to be close to the worst-case scenario for maintaining the tags. In [BCDFZ] the analysis is done in terms of log n instead of w.

This basic idea has been implemented in the library, but is not the compilation default. It is very simple and has very low overhead on operations that does not cause tag redistributions. In the worst case, all tags must be redistributed, which means the complete list will be traversed, which is bad for locality of memory references.

To get to O(1) amortized cost per list update of tag maintenance there is a standard two-level grouping scheme: Use the simple scheme to maintain tags on groups of consecutive list nodes and use a second tag inside each group. If the number of groups are kept at O(n/w), say by making at least every other group be of size Ω(w), then the amortized cost of maintaining the top level tags will be O(1). It is not hard to see that putting an extra O(1) charge on insertions will pay for splitting and renumbering groups when we miss room in the low level tags; and an extra charge on deletions will pay for joining groups when a group that gets below the Ω(w) threshold has a neighbor already below. In total we get an O(1) amortized cost per list update of tag maintenance at the cost of a much more complicated algorithm. Performance measurements indicated that this scheme nevertheless performs better at large list sizes so we made it the compilation default.

The tag groups are implemented as the nested class TagGroup:

```
class TagGroup
{
     int tag, count;
     Node first, last;
}
```

The first and last fields enable finding all members of the group and indirectly following the chain of tag groups. Tag maintenance is done by utilities gettaggroup, settag, removefromtaggroup, redistributetaggroups. The ordering of nodes can be computed by a method on Node:

```
class Node {
     …
     bool precedes(Node that)
     {
         int t1 = taggroup.tag, t2 = that.taggroup.tag;
         return t1 < t2 ? true : t1 > t2 ? false : tag < that.tag;
     }
     …
}
```

As might be guessed from the code snippets in the previous section, the source code for LinkedList<T> contains all the tag maintenance code and data, but the tag maintenance code will not be executed because it is guarded by tests of the maintaintags flag field. This is so because HashedLinkedList<T> is implemented as a subclass of LinkedList<T>. The main disadvantage is the substantial (typically 67%) hit on the size of a node in LinkedList<T>.

6.2.3.3 HashedLinkedList<T> class

The complexity of the HashedLinkedList<T> class is characterized by having linear time operations by index and amortized constant time operations by item.

As noted, HashedLinkedList<T> is a subclass of LinkedList<T> with the maintaintags flag field set to true. Roughly half the public methods is overridden, the rest works because the utility methods is overridden by versions that use the hash index.

6.2.4 Heaps

For heap data structures (implementing priority queues), the library contains two implementations, the classical binary heap and the less well-known interval heap of whomever, [LW], which unlike the binary heap supports both maximum and minimum operations.

The interval heap was chosen in favor of another simple two-sided heap, the MinMax heap. The choice was based upon the results in a couple of reports from the Copenhagen STL ([CSTL]). We have also refrained form implementing binomial (or Fibonacci) heaps, whose main virtue is to support efficient merging of two heaps.

Note that the red-black trees can also be used as priority queues, with the same complexity for the queue operations, but considerably larger constant factors (chapter 7).

6.2.4.1 BinaryHeap<T> class

The complexity of the binary heap is characterized by logarithmic complexity Add, DeleteMax operations.

The interval heap performs almost as well as the binary heap so we expect to remove the binary heap from the library, since it violates the requirement of always having real implementations of exported methods.

In a binary heap, the items are organized in a balanced binary tree with the leaves at maximal depth flush left. The heap property is that each item is at least as large as its (at most) two children. We use the standard heap layout of placing the items in a (dynamic) array so that the parent-child relationships are given by the simple arithmetic formulae for the array indices:

parent = [child/2], left_child = 2*parent, right_child = 2*parent+1.

The implementation is based on the heapify(i) method, which starting at the item at array index i pushes that item downwards in the tree by swapping with a child as long as a heap property violation is seen.

We employ the standard O(n) implementation of AddAll, even though some authors report that the naïve O(n*log n) AddAll implementation of calling Add repeatedly may perform better on modern machines because of much better locality of reference. We have not tested that issue.

6.2.4.2 IntervalHeap<T> class

The complexity of an interval heap is logarithmic time Add and DeleteMin/-Max operations.

The interval heap is organized by having pairs of items instead of individual items be laid out in balanced binary tree like the one of a binary heap. The last pair is a singleton if the size of the heap is odd. The pairs are called intervals and the heap property is that each interval encloses all the intervals below it in the tree (i.e. its child intervals).

Our implementation uses a value type to represent an interval:

```
struct Interval
{
   T first, last;
}
```

The interval heap has a dynamic array of Interval structures laid out in the same fashion as the binary heap. The code is built around four utility methods: heapifyMin(int i), heapifyMax(int i), bubbleUpMin(int i) and bubbleUpMax(int i).

The heapifyMax operation works like classical heapify working downward in the tree of "last" fields. At each level below the top after a swap of a parent and child, we may have to swap the first and last field of the child before proceeding downwards. Method heapifyMin is symmetric.

The bubbleUpMax method moves an item from the bottom upwards along either last field by swapping child and parent as long as a violation of the interval heap property is seen. No swapping of first and last is needed.

Delete operations deletes from the top interval, moves the final item of the heap to the empty place at the top and then runs the relevant heapifyMxx method. Add may expand the array and then adds the item to the final interval or as the first field of a new final interval, then runs bubbleUpMxx as relevant. The AddAll is implemented as repeated Add operations and therefore has complexity O(n*log n), which probably is sub optimal (in a pure RAM computational model).

6.2.5 Search trees

We have implemented some variations of the well-known red-black trees. The basic construction is taken from Tarjan [Tarjan1] with extensions for partial persistence as described in [DSST]. The operations are done in so-called bottom-up style and we have no parent pointers in the tree nodes. Compared to the tree collections of Java, the main differences are the support for persistence and the support for fast lookup by index.

As mentioned earlier, splay trees would not fit into the library. It would be interesting to try some higher degree trees (B-tree like) to compare performance with the red-black trees.

Note that we could base an implementation of IList<T> on this tree implementation by using just the index related lookup and update methods. This would give a list collection mediating between dynamic array lists and linked lists in performance.

6.2.5.1 RedBlackTreeSet<T> class

The complexity is characterized by having lookups and single item update operations of logarithmic complexity in the size of the tree. As long as we do not use persistency (the Snapshot operation), the space use of the tree is linear in its size. When using persistence, the space use is linear in the number of single item update operations since the creation of the tree.

The RedBlackTreeSet<T> class has several nested classes:

- Node, the basic building block
- Enumerator, SnapEnumerator used for enumerators (ordinary and of snapshots)
- Range, Range.Enumerator, Interval for results of range and interval queries
- SnapData, for keeping track of which snapshots are alive.
- Feature a debugging aid (to be removed)

A red-black tree holds the following data:

```
class RedBlackTreeSet {
     IComparer<T> comparer;
     Node root;
         //Stack for iterative internal update utilities:
     int[] dirs = new int[2];
     Node[] path = new Node[2];
         //For persistence:
     bool isSnapShot = false;
     int generation;
     SnapData snapdata;
     int maxsnapid = -1;
     …
}
```

The tree object holds a single reference to the real tree structure built of Node objects. Persistence will be discussed in next section. The stacks for iterative internal update utilities are needed because we do not have parent pointers in the nodes, partly because they would make the persistence much harder. The Node class has this content:

```
class Node
{
     bool red;              //color
     T item;
     Node left, right;
     int size;              //of subtree
         //For persistence:
     int generation;
     int lastgeneration;
     Node oldref;
     bool leftnode;
}
```

The subtree size field enables fast (logarithmic time) index to item and item to index operations.

The most fundamental parts of the implementation are the private methods addIterative, removeIterativePhase1 and removeIterativePhase2. As the names suggest they are iterative instead of recursive, which seems to enable about 10% better performance. The first of the remove methods is the part that walks down the tree to find the node to remove. The second remove method is used as the backend of several public methods that removes items but have different ways of finding which Node to remove (by item, min, max, by index).

During implementation, many minor variations was tried and compared. Apart from recursive vs. iterative basic methods, preprocessor symbols allows to select to

- Maintain subtree sizes or not (for indexing)
- Maintain exact tree height
- Maintain exact black height (red-black rank)
- Maintain max and min (for O(1) FindMin and FindMax)
- Maintain unique ids on Nodes.
- Enable path copy persistence, node copy persistence or neither.

Some of these are mainly debugging aids; some were made to measure how expensive certain features would be. Unfortunately, the alternatives make the code much less readable.

Two public methods have suboptimal implementations. The various RemoveRange methods and the RemoveInterval method works by simply removing all items in a loop hence could use time proportional to n*log n in the worst case. It is possible to implement operations on a red-black tree that in O(log n) time splits a tree at a specific Node into two trees or joins two trees with all items in one less than all items in the other. Such operations could speed up the RemoveXxx operations considerably. It is not trivial to combine them with persistence, though.

The AddAll and AddSorted implementations have the same problem we have seen before, that the best algorithm depend on the size of the tree and the number of items to add.

6.2.5.2 Node copy persistence

This algorithm is from Driscoll et al, [DSST]. The persistence is governed by an integer generation id field in the tree, incremented on each call to Snapshot(). The generation field of new Node object gets its value from the tree. Recall that this is partial persistence – the snapshots are historic documents, not quick mutable clones. The snapshot itself is a shallow clone of the tree object having the old generation id and the ReadOnly property and the isSnapShot flag set to true. For very long running programs (or specially crafted test cases), the 32 bit signed generation id may wrap around, which will not be handled gracefully by the library.

We will now describe how the Node fields are used for defining a specific generation snapshot, i.e. how to enumerate a specific generation of the tree. As noted, the generation field of a Node is equal to the generation id of the tree when it was created and the Node will never be encountered during enumeration of an older snapshot. The item of a Node encountered during enumeration of a snapshot will always be valid. The color (red field value) is not valid for the snapshot; in fact, the color is only needed for updates, i.e. the live tree. If a Node with generation field g0, and lastgeneration field g1 is encountered when enumerating a newer snapshot of generation g2, where g1<g2, then both child references in the Node are valid. If the snapshot is older, g0<=g2<=g1, one of the child references in the Node will not be valid; instead the valid child reference is in the oldref field and which child is shown by the leftnode field.

How updates to the tree are modified by persistence will now be explained (we assume the reader is familiar with updates in non-persistent red-black trees). When we are about to update a Node whose generation field is not newer than the newest live snapshot – given by the maxsnapid field of the tree – the following procedures are followed. If the update is a color change, just do it. If the update is a change of the item, make a copy of the node, update the item in the new node and update the reference in the parent to point to the new node. Note that the new node will have the generation field of the tree we are updating. If the update is a change of a child reference and the lastgeneration field of the Node is still -1 we update lastgeneration to maxsnapid, set the leftnode field to true if it is the left child we are updating, copy the old child reference to the oldref field and finally update the live child reference. If, finally, we must update a child reference in a Node that has already had made one child reference update, we copy the node to a new one with new generation id and update the child pointer in the parent. In the last situation, if the child reference we are updating is the child that was updated before and the result of the old update cannot belong to any live snapshots because the tree's maxsnapid <= lastgeneration of the Node, we just update the child reference instead of copying the node.

It should be clear from the explanations that the procedure is correct and does not influence enumerations and lookups in the tree and snapshots by more than O(1) work per node encountered. Thus single item operation will still use time O(log n). We now explain the crucial fact that the procedure will only produce O(1) amortized new nodes per update operation on the tree. Note that a single update operation on the tree can result in cascading node copies all the way to the root. But since the cascade will stop when we use an oldref slot or reach the root, each update operation on the tree can result in at most O(1) oldref slots being filled. Moreover, a Node will only be copied once since after the copy it will no longer be part of the live tree. Now, if we have performed m updates operations on the tree since its creation, there can be at most O(m) Nodes with filled oldref slots. A Node with an unfilled oldref slot either has been created directly by an insert operation on the tree or is the single copy of a Node with a filled oldref slot. In total we have at most O(m)

Nodes of either kind, which means exactly that we will only produce O(1) amortized new nodes per update operation on the tree.

The central utility methods for persistency are the left(node) and right(node) methods of RedBlackTree<T> that navigate the tree/snapshot according to the rules above; and the update(…) and CopyNode(…) methods on Node that perform the update or copy on a single node. Calls to these utilities are dispersed in generous quantities all over the code.

The maxsnapid field holding the newest live snapshot is maintained in the following way. When a snapshot is created, it is registered in a SnapData object and the snapdata and maxsnapid fields of the original tree are updated. The storage element in the SnapData class is a red-black tree itself. When a snapshot is disposed, either by an explicit call to Dispose() or by the garbage collector calling the finalizer of the RedBlackTreeSet<T> class, that snapshot will deregistered for the SnapData object and maxsnapid updated if relevant. In particular, maxsnapid may be reset to -1 if there are no live snapshots. This means that if we take a single snapshot, keep it alive for some time, and then dispose it, we may avoid copying the whole tree if we only make a few updates while the snapshot is alive.

It would be possible, but expensive, to make a more thorough cleanup of Nodes that becomes unreachable when a snapshot does. We do not because the typical usage modes would be

- Make a snapshot, enumerate it while doing updates, and then dispose the snapshot.
- Build a data structure where we need all the snapshots until we just forget the whole thing.

Both are handled well by the current implementation. Anyway, the library has a space leak. With three references per tree node, we could end up using space super linear in the number of live items.

Node copy persistence is activated at compile time by defining the symbol NCP.

As hinted to in the previous section, the source code also contains an implementation of path copy persistence – similar to typical functional tree implementations. The problem is that it copies ~log n Nodes on each update operation, not just O(1) amortized, which makes it require much more space to handle large trees with frequent snapshots. Path copy persistence is activated at compile time by defining the symbol PERSISTENT.

6.2.5.3 RedBlackTreeBag<T> class

The complexity is characterized by logarithmic complexity lookup and (single-item) update operations.

The implementation is a slight variation of the tree set implementation, with an extra count field in each tree node. In fact, the source code for the set implementation also holds the bag code, which is selected by defining the BAG preprocessor symbol. There are a few methods with separate implementations, and the bag versions of FindOrAdd and UpdateOrAdd are wrong.

6.2.5.4 RedBlackTreeDictionary<K,V> class

The complexity is characterized by logarithmic complexity lookup and (single-item) update operations.

The implementation is almost trivial, based on the DictionaryBase<K,V> base class with trivial implementations of the extra predecessor and successor operations of ISortedDictionary<K,V>.

There is a Snapshot method, routed directly through to the inner tree set. This is a violation of library design principles, because there is no interface for a persistent dictionary.

6.3 Support classes

We do not show support classes as diagrams – they would be quite trivial.

6.3.1 Item hashers

The library contains a number of helper classes for constructing item hashers:

- DefaultReferenceTypeHasher<T>,
- DefaultValueTypeHasher<T>,
- IntHasher

and for constructing hashers for items of collection type:

- HasherBuilder.ByPrototype<T>,
- HasherBuilder.SequencedHasher<S,W> where S: ISequenced<W>,
- HasherBuilder.UnsequencedHasher<S,W> where S: IEditableCollection<W>.

The generic DefaultReferenceTypeHasher<T> and DefaultValueTypeHasher<T> are trivial implementation of IHasher<T> based on calling the GetHashCode and Equals methods inherited from System.Object. There are two different default classes because the equality code is different. For value types we can just call "i1.Equals(i2)", while for reference types we need to do

```
(object)i1 == null ? (object)i2 == null : i1.Equals(i2).
```

It does not seem to be possible to fuse the two classes without incurring a runtime penalty of e.g. checking a field value. Unfortunately, the generics functionality (on the test platform) does not allow forcing the parameter T to be of value type or reference type at compiletime, and so we rely on a parameter instantiation time check in Debug builds to test those conditions.

The "IntHasher" class is simply DefaultValueTypeHasher<int> with explicitly inlined int hash function and equality test. There ought to be similar classes for other builtin value types like double, long etc.

The HasherBuilder.ByPrototype<T> uses reflection on T in the "Examine" method to create a default hasher of one of the above types unless T has an exact interface type of one of the following kinds: ISequenced<W> or IEditableCollection<W>. In the latter cases, the (Un)sequencedHasher classes is used to construct item hashers that use the collection hash functions described in chapter 5. This implies that the default item hasher for, say, HashSet<ISeqenced<int>> will be the sequenced collection hasher, while the default item hasher for HashSet<LinkedList<int>> will be the default reference type hasher (for LinkedList<int> items).

We can directly construct the sequenced collection hasher for a class, C, implementing "ISequenced<T>" for some T by the expression "new SequencedHasher<C,T>()" and similarly for unsequenced collection hashers. Note that the hasher used for the T equality tests performed while comparing two collections of T will be the IHasher<T> hasher of (one of) those collections!

The library source code contains inactive code for alternative ways of building the collection hashers: by using reflection and runtime invocation of reflected methods and by using reflection and runtime code generation. The first procedure did work, but would to lead to terrible performance. The second procedure could not be made to work on the available alpha release platform because of problems with runtime code generation for code using generics.

6.3.2 Item comparers

The library contains a few classes

- NaturalComparer<T> where T: IComparable<T>
- NaturalComparerO<T> where T: System.IComparable
- IC
- ComparerBuilder.FromComparable<T>

The “natural” comparers are trivial wrappers of the “Compare” methods of the generic and non-generic comparable types. The IC class is NaturalComparerO<int> with explicit inlining. There should be similar classes for double etc. The Examine method of ComparerBuilder.FromComparable<T> will choose one of the first three based on the instantiation of T or throw an exception if T is not comparable of either kind.

6.3.3 Base classes

Corresponding to the central spine of collection interfaces, IEnumerable<T>, ICollection<T>, IEditableCollection<T>, ISequenced<T> the library contains a string of helper classes meant to be used as base classes in implementations of the interfaces. In addition, there is one helper class to be used as base a class of dynamic array classes:

- EnumerableBase<T>,
- CollectionBase<T>,
- EditableCollectionBase<T>,
- SequencedBase<T>,
- ArrayBase<T>.

They were constructed when most of the real collection classes had been coded by extracting common operations and data fields – unifying the implementations at the same time.

Only the two first helper classes implement the corresponding interface. The other ones do not since it would just lead to a large number of abstract method definitions. We do not need to implement the interfaces in the base classes for the following reason. If C has B as base class, implements interface I and a certain method signature “X f(Y y)” is defined in I and implemented in B, then the two methods will be identified by the compiler even if B does not implement I. The two first helper classes **do** implement the corresponding interfaces because we need the interfaces in the class implementations.

Some of the common data fields in these classes are private and therefore fine from an engineering viewpoint. However, the ”isReadOnly”, ”stamp”, ”size” and ”itemhasher” fields of ”EditableCollectionBase<T>” and the ”array” and ”offset” of “ArrayBase<T>” are all protected. That is problematic because a user subclassing the collection classes built on the base classes could subvert invariants like “the size field contains the number of items in the collection”. There may be a way around this problem, by giving read-only access to the fields through a “protected” property and write access through an “internal” property that would only be accessible by library code, but we have not considered this in detail.

Most of the code in the base classes are obvious, e.g. a default implementation of “ToArray” in “EditableCollectionBase<T>”. In the following, we mention the few less obvious ones.

EnumerableBase<T> has a static method called “countItems” which count the number of items in a collection by enumerating through it, unless the collection implements “ICollection<T>”, in which

case the Count property is used. This potentially expensive method is used a couple of places in the library where we need to know the size of a collection we only know implements “IEnumerable<T>” (supplied as a parameter to some method).

EditableCollectionBase<T> implements the static methods ComputeHashCode and StaticEquals and the protected instance methods unsequencedhashcode, unsequencedequals which real collection classes will use for implementing the interface specific unsequenced collection hasher methods “IEditableCollection<T>.GetHashCode” and “IEditableCollection<T>.Equals”. The base class has private fields for caching the collection hash code.

SequencedBase<T> has parallel functionality for sequenced collection hasher support.

ArrayBase<T> contains basic functionality for dynamic arrays in the protected expand and insert methods working on the protected array field. It also has a nested class “Range” with a complete implementation of directed range/interval enumerators for dynamic array based collection classes.

6.3.4 Dictionaries

For dictionaries, we have the following simple entry-level classes:

- KeyValuePair<K,V> : the type of dictionary entries
- KeyValuePairHasher<K,V> : default entry hasher using only keys
- KeyValuePairComparer<K,V> : default entry comparer for sorted dictionary comparing keys only.

The DictionaryBase<K,V> class is a complete implementation of a dictionary in terms of a set collection. All the methods are just small wrappers around call to operations on the internal set collection object.

To support the Keys and Values properties, DictionaryBase has the straightforward nested classes KeysCollection and ValuesCollection.

6.3.5 Other support classes

EnumerationDirection is an enum class with two values: Forwards and Backwards, used for describing the direction of enumeration of a directed collection relative to the original collection. We could have relied on a Boolean and naming the relevant method IsEnumeratedForwards instead of Direction, which seems just as good.

TestedAttribute is a custom code attribute that can be attached to methods and properties to track which operations are sufficiently covered in the regression test cf. chapter 4. The attribute can be applied multiple times to a method and has an optional string argument that may be used to reference a test case, but these features have not been used.

C5Random is a class implementing the recent strong random number generators CMWC by George Marsaglia ([Marsaglia]). There is no interface in CLI for random number generators, but we have made C5Random a sub class of System.Random, which allows it to be used where System.Random is used.

6.3.5.1 Sorting

The standard library includes support for sorting arrays that we need in a few places. The implementation seems to be a fully recursive median-of-3 quicksort. While it seems to perform very well, it is not of optimal asymptotic complexity (O(n^2) vs. O(n*log n)). In fact, measurements presented in chapter 7 seem to indicate that the CLI standard array sort is vulnerable of the “killer” sequences of [Musser].

Therefore, we have chosen to implement a version of IntroSort of Musser [Musser]. This is a modified version of median-of-3 quicksort with the following modifications. The recursive partitioning in quicksort at most proceeds to a depth proportional to the logarithm of the size of the array. If this depth is reached, the corresponding partition is sorted by heap sort. Moreover, if a partition with size below a constant value is seen, no further partitioning is done and the partition will be sorted by insertion sort. Since each partition depth is of at most linear complexity, heap sort is asymptotically optimal and the insertion sort will be of total at most linear complexity, the complete IntroSort will be asymptotically optimal.

The implementation is in the class Sorting and its nested generic class Sorter<T>. During implementation, many attempts were made to “beat” the standard library quicksort on random data. It was attempted to change the recursive partitioning to iteration+recursion and pure iteration with an explicit stack. Moreover, several attempts was made to “optimize” the code, typically making it more complicated trying to take advantage of special cases. The results were not completely satisfactory; the standard library is slightly (a few %) faster for random data on large arrays.

6.3.5.2 ReadOnly wrappers

Finally, there are 12 generic classes named ReadOnlyXxxx, where IXxxx ranges over most of the collection interfaces. These are all simple wrapper classes that have a single constructor taking an argument of the corresponding interface type and implement that interface by calling through to non-update operations directly and throw an exception on update operations. The only slightly non-trivial task in the implementation is to remember to protect return values of collection type, because an IEnumerator returned from the original collection might actually be e.g. the original collection itself and expose update operations if not wrapped.

7 Evaluation

7.1 Functional test

Organization of unit tests: The tests are put in a separate top-level namespace "nunit" in a separate VS.Net project and further grouped by a couple of namespaces levels. The names of test methods are typically constructed from the most prominent operation tested in that method, but the correspondence between test methods and tested operations cannot be traced by test method name.

Test tracing: The library contains a "Test" custom attribute, which is used to keep track of which public methods and properties are covered sufficiently by test cases. During implementation, the Test attribute is only put on a method or property with seemingly sufficient test coverage of correct and exceptional usage. A special program was written to walk through the library via reflection and export the types and their members together with custom attributes to a database, where the completeness of the test coverage could be analyzed by various dimensions and granularity.

Note: the tracking does not cover static methods, inherited methods or explicit method invocation. In practice, we made test cases for these members also, but we have no mechanism to assess the completeness of the covering.

The selection/construction of test cases follow usual folk guidelines: there are first of all simple black-box tests of the normal behavior in simple cases, assumed corner cases are added (operations at ends of lists etc.) and similar cases where one could expect something to go wrong in a careless implementation. Only an informal approach was used to make it plausible that all code paths were covered. In the cases of red-black trees and hash tables, a more systematic approach was used to make sure all of the special cases were encountered. For the red-black trees, this is only true for the non-persistent case. In the persistent case, there are too many different combinations of node color pattern and node generation pattern to make a completely systematic test. When a bug was uncovered by other means (e.g. by the random tests mentioned below), an attempt was made to make a unit test case provoking the bug in order that it would not reappear without detection.

To complement the black-box flavor of the unit tests, the collection classes all have a "Check()" method that performs an internal consistency check on its private data. The typical test method would contain a number of ordinary collection operations (with assertions on result if non-void) interspersed with calls to assert that "Check()" returns true and calls to a utility method that checks the complete contents of the collection at this point.

As an extra correctness test, the "Check()" method has been used in connection with random operation sequences. To be useful, such tests makes tens of thousands of random adds and removes and calls Check() in between. Because the check operation may be quite slow, this takes a lot of time and is not suitable to include in the standard regression test suite.

The following lists the public methods not covered according to database:

- FindOrAdd and UpdateOrAdd on RedBlackTreeBag (known to be wrong)
- Shuffle on (hashed) arrays and linked lists (6 method in total)
- All 5 methods on C5Random
- 3 ToString on private classes (false negative)

Note: the Check() methods are used extensively in test cases but there are no test cases designed specifically for them.

There are about 900 individual test case methods and a full regression test takes about 60s on the development machine (a 433 MHz Celeron).

The tests for two classes with the same interface **and** the same value of NoDuplicates are the same, or almost the same. The same can be said of the tests for the operations in the common interfaces of several classes with the same set/bag semantics. While implementing the second of such classes, we just bulk copy the test cases, and update the class names in the copy. When a bug in one of the classes was uncovered by some other means (perhaps a random test), a test case exposing the bug would be added, in some cases only to the offending class. In that way, the collection of test cases for the two classes could drift apart. It would be nice, if the code of the test cases could be shared on an interface-by interface basis, which seems to require that we generate real test cases programmatically from interface ones.

7.1.1 Partial conclusion

We are confident that there are few unknown bugs in the library due to good test coverage.

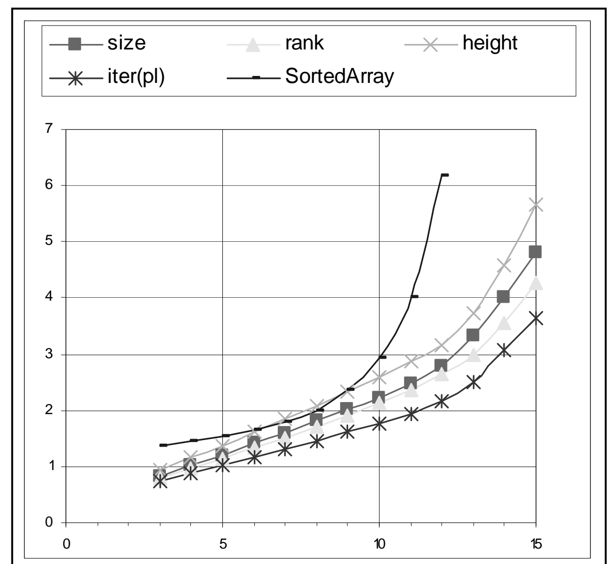
7.2 Performance tests

We will not give a complete description of all performance tests we have done, just present some representative results. As noted in chapter 4, the coverage of our testing is weak on non-integer collections and on comparison with other libraries.

Note: In all the micro benchmark graphs presented are “full saw tooth” benchmarks of collections of integers. The horizontal axis shows the base-2 logarithm of the maximal size of the collection. The vertical axis will show a value proportional to the running time per operation (the precise unit varies).

7.2.1 Implementation alternatives for red-black trees

The first graph shows a few of the alternatives we have tried for red-black trees. The low graph marked “iter(pl)” is our iterative implementation with as few features as possible. The three graphs marked “size”, “rank” and “height” shows the result of supporting one of the following: subtree sizes (i.e. indexing), precise height and red-black rank. The standard compilation setting for the library correspond to “size”, about 10% slower than the plain “iter(pl)”.

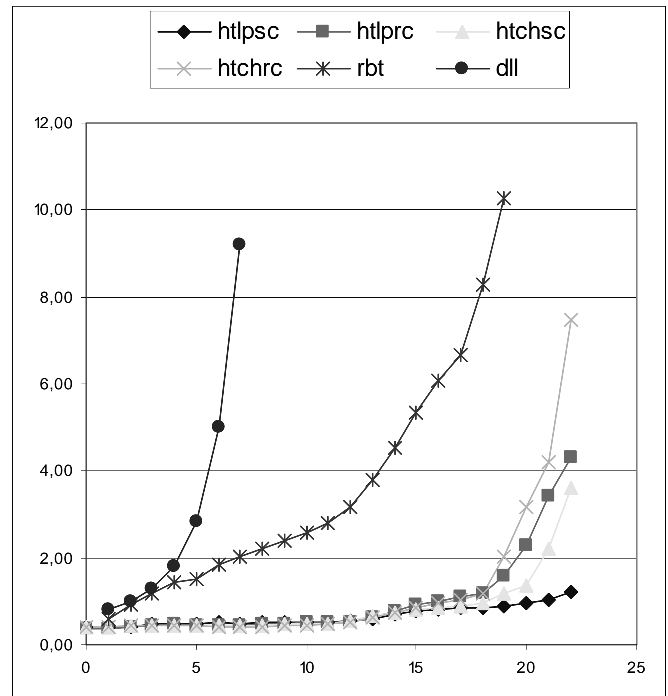


The graph for SortedArray shows clearly how the explosion in running time (~n) compared to the tree data structure.

The bends on the graphs around a tree height of 12 seems to correspond to the situation the trees no longer fit completely in the second level cache.

7.2.2 Implementation alternatives for hash tables

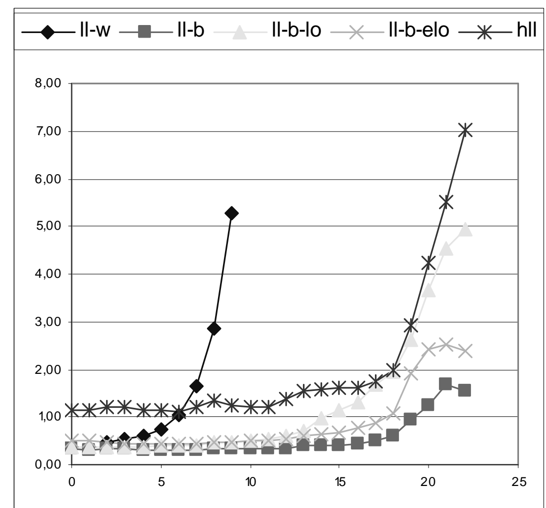
The graphs marked “ht” represent the four hash table types:
lp is linear probing,
ch is chaining
sc is value type bucket
rc is reference type bucket.



On the figure they are very close and close to constant until what seems to be memory management effects from 2^18 items and up. The best performing at large sizes is linear probing with value type bucket, which has become the standard in the library.

For comparison, we also show result for the linked list (dll) and red-black tree (rbt). They both behave, as one should expect.

7.2.3 Linked lists and sequence order

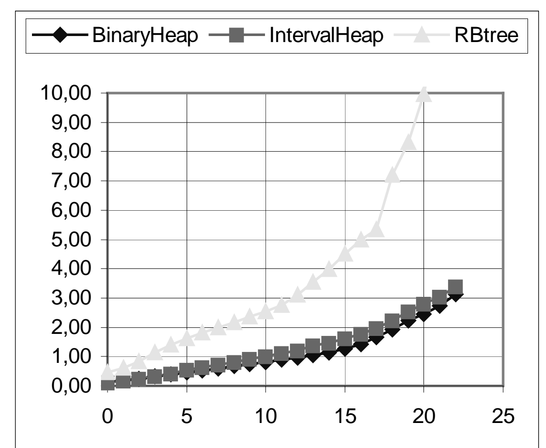


The graphs show the performance of linked lists. The “ll-w” case is the worst case (the removes have to traverse the whole list to find the target), the rest of the cases are for the best case.

The “ll-b-lo” uses the simple sequence order algorithm, the “ll-b-elo” uses the complicated, but asymptotically better one and “ll-b” uses no such algorithm. As expected, the complicated one behaves better than the simple one at large size, but are not free.

The “hll” case is a hashed linked list (without sequence order algorithm). The overhead on these operations roughly triples the running time, but before we implemented the combined operations like UpdateOrAdd to support faster dictionary operations, the overhead were closer to ten-fold.

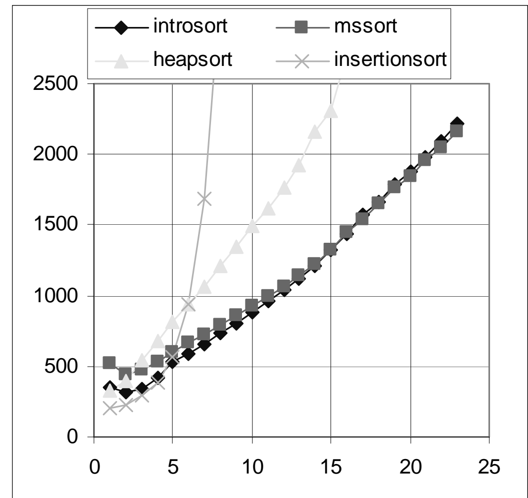
7.2.4 Priority queues



The graphs show the performance of the priority queue operations on trees and the heaps implemented.

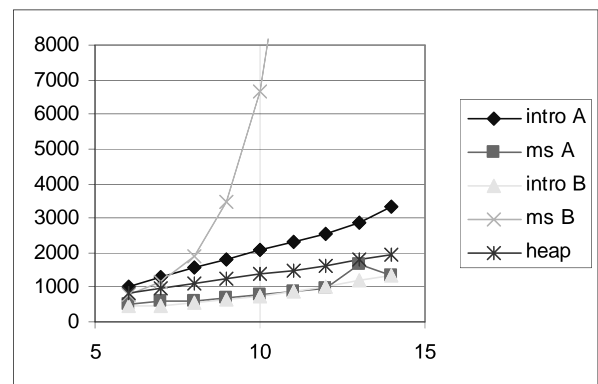
The main conclusion is that the heaps really are much faster than the trees for this purpose and that the interval heap is so close to the classical binary heap that we can do without the latter.

7.2.5 Sorting



The first graph compares the time per array element for sorting an array of random integers with our own implementatios of heapsort, insertion sort and intro sort and the standard Array.Sort class (marked mssort). It can be seen that for large arrays we are quite close to the standard sorting. (The horizontal axis shows the base-2 logarithm of the array length).

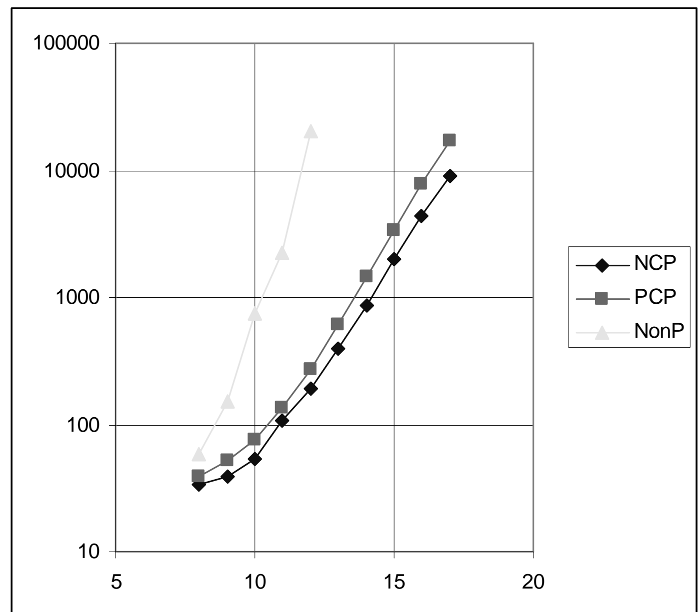
Insertion sort is of course hopeless for anything but very short arrays. Heap sort shows O(n log n) asymptotics with a higher constant than intro sort.



We have also tried to test the “killer” sequences of [Musser]. The “A” graphs use the exact killer sequences; the “B” graphs use them reversed.

It can be seen that the quicksort sorting of Array.Sort does not seem to handle the “B” sequences very well. Our intro sort behaves better, but worse than heap sort as should be expected, since the killer sequences will make the first couple of dozen partitionings be almost useless, after which intro sort will let heap sort take over.

7.2.6 A geometric application example, a case for persistence



The final test is the only non-synthetic one we have made. It is one of the standard examples of the usefulness of persistent trees ([DSST]).

The problem is, given a set of line segments dividing the plane into cells, create a data structure that can quickly answer to which cell a given point in the plane belongs.

The solution is to have a persistent search tree of line segments sorted by y-coordinate, with the line segments added and removed according to a horizontal sweep – taking a snapshot at each x-coordinate of an endpoint. Looking up a point in the plane then first selects a snapshot by x coordinate and then searches the snapshot for the y coordinate.

The graphs show the results of a nasty example, where there is only a couple of updates in between each snapshot, but the trees hold most of the time at least half the total number of line segments.

The x coordinate of the graph is the base 2 logarithm of the total number of edges. Note that the y-axis has a logarithmic scale.

It is clear that the non-persistent case (where snapshot is a full clone) is only useful for very small examples, and node copy persistence (NCP) clearly outperforms path copy persistence (PCP).

7.2.7 Partial conclusion

We have measured the relative performance of a number of implementation alternatives in order to make an informed choice. We have also made some comparisons of the performance of different classes.

The results are much weaker when it comes to assessing the influence of the item type and comparisons with other libraries.

7.3 Extensions, variations and missing things

Here we sum up things we should have done or would have liked to do. Appendix C contains a more detailed list of TODO items for coding. We try to partition the issues according the words “must”, “should”, “nice”.

7.3.1 Must

This refers to issues that must be resolved before using the library in production.

First, known wrong semantics or bad complexity must be repaired. Two of the combined operations on RedBlackTreeBag<T> have wrong semantics. Some of the implementations of RemoveAll, RetainAll etc. are suboptimal in some classes. The collection hasher implementations do not follow the design.

The interfaces and classes reference manual does not show inherited methods making it misleading. The class descriptions in the reference manual are too short and miss thorough performance and usage notes.

7.3.2 Should

Extend user’s guide.
Make collection classes clonable and serializable.
Remove BinaryHeap

7.3.3 Nice

Improve the layout of the reference manual.
Implement operations like join and split for the collections.
Implement some form of perfect hashing
Improve views and/or support fingers
Implement “wildcard adds” – perhaps awaiting support from CLI
Get to really understand equality in connection with collections.

7.3.4 Partial conclusion

We have a few critical, but well-described coding tasks to do and a larger editing task concerning the manual.

7.4 Platform anomalies

We end this chapter with a few short remarks on our experience with the platform. Recall that the VS.Net used was an alpha release

We were taken by surprise by a case of slow static field access in the CLR. For debugging purposes, our tree implementation may give each tree node a unique id, initialized from a static nextid field. It turns out that enabling this feature doubles the running time and the culprit is the accesses to nextid. We did not examine this further but avoided static fields afterwards.

We discovered the hard way during the implementation of the sorting routines the difficulties of fine-tuning C# code to squeeze out the last bit of performance. In one case, it was reproducibly more efficient to copy a method parameter to a local variable to use in a loop, but in very similar situations, it could be more efficient to use the parameter as loop variable. [Noriskin] explains that it is difficult when modifying C# code to ensure that the C# compiler will generate intermediate code that is easier for the JIT compiler in its turn to generate faster native code for.

We were hindered in using runtime code generation (RTCG) in a couple of interesting cases because that functionality did not work with generic classes. The RTCG functions would generate invalid code containing badly formatted type names.

We had some problems with crashes of VS.Net, but normally the environment would discover something was wrong and warn so one could save files and restart. Another strange problem was that one should open the properties window before opening the first project if one wished to use certain, seemingly unrelated menus (the solution was announced in a newsgroup). None of this is sensational for an alpha release.

8 Conclusion

We have made a library of generic collection classes for .Net.

The functionality of the library matches or surpasses the Java collection classes in range of data structures implemented and the level of functionality of the implementations.

A comprehensive regression test suite with good coverage assures trust in the correctness and reliability of the library, except for a few known errors. The regression test suite also eases revision of the library.

The usability of the library we have attempted to ensure by following a set of rules outlined in chapter 3. Unlike Java and .Net, the collection classes are designed for programming to interfaces and not implementations. There is a substantial amount of reference and other documentation, which needs some revision to be completely satisfactory though.

Good asymptotic performance of the library has been designed into it by employing first class data structures. The implementation has been tuned with respect to internal coding alternatives, but has not been measured extensively against competing or comparable libraries.

Portability of the library and suitability for a range of programming languages will rely on the same properties of the .Net platform.

To aid the reuse of this work for the benefit of programmers, it has been put under an MIT/X11 style open license.

Lists of tasks for further work has been laid out in chapter 7 and the TODO list of the appendix.

Literature

[BCDFZ] M. Bender, R. Cole, E. Demaine, M Farach-Colton and J. Zito, *Two Simplified Algorithms for Maintaining Order in a List*, Proceedings of the 10th Annual European Symposium on Algorithms (ESA 2002), Lecture Notes in Computer Science, volume 2461, Rome, Italy, September 17-21, 2002, pages 152-164

[BF] P. Beame, F. Fich, *Optimal Bounds for the Predecessor Problem*, STOC'99

[CE] K. Czarnecki and U. Eisenecker, *Generative Programming*, Addison-Wesley 2000

[CLRS] T. Cormen, C. Leiserson, R. Rivest and C Stein, *Introduction to Algorithms*, MIT Press 2001

[Cook] William R. Cook, *Interfaces and Specifications for the Smalltak-80 Collection Classes*, ACM OOPSLA 1992, p 1-15

[CSTL] Copenhagen STL, web page available march 2004 at: http://www.cphstl.dk/WWW/index.php

[dotgnu] Main dotGNU we page: available march 2004 at http://www.dotgnu.org/

[dotNet] Main Microsoft .Net web page: available march 2004 at http://www.microsoft.com/net/

[DSST] J. Driscoll, N. Sarnak, D. Sleator and R. Tarjan, *Making Data Structures Persistent*, Journal of Computer and System Sciences, Vol. 38, No. 1, February 1989

[ECMA] CLI standard, available march 2004 as

http://www.ecma-international.org/publications/standards/Ecma-335.htm

[Gray] Jan Gray, *Writing Faster Managed Code: Know What Things Cost*, June 2003, on Microsoft Developer Network, http://msdn.microsoft.com/

[JGL] Java Generic Library, info available march 2004 at http://www.recursionsw.com/products/jgl/jgl.asp

[Knuth] Donald E. Knuth, *The Art of Computer Programming 1-3*, Addison-Wesley 1997

[Lausen] S. Lauesen, *Software Requirements, Styles and Techniques*, Addison-Wesley 2002

[LW] J. van Leeuwen and D. Wood, *Interval Heaps*, The Computer Journal, volume 36, number 3, 209-216, 1993

[Marsaglia] George Marsaglia, *A pseudo-random number generator of type `complimentary multiply with carry' (CMWC)*, Communications of the ACM 46, 5 (May 2003) 90-93;

[mono] Main Mono project web page: available march 2004 as http://www.go-mono.org/

[MR] R. Motwani and P. Raghavan, *Randomized Algorithms*, CUP 1995

[Musser] David R. Musser, *Introspective Sorting and Selection Algorithms*, Software Practice and Experience, Vol. 27, No 8, p983-993, 1997

[Noriskin] Gregor Noriskin, *Writing High-Performance Managed Applications: A Primer*, June 2003, on Microsoft Developer Network, http://msdn.microsoft.com/

[Nunit] Main Nunit web page available march 2004 at: http://sourceforge.net/projects/nunit

[PR] R. Pagh and F. Rodler, *Cuckoo Hashing*, Lecture Notes in Computer Science 2161 (2001) 121ff

[PS] Generic C# Sample Programs, available march 2004 as http://www.dina.kvl.dk/~sestoft/gcsharp/

[SD] D. Sleator and P Dietz, *Two Algorithms for Maintaining Order in a List*, Proc. 19th Annual ACM Symp. on Theory of Computing (STOC'87), ACM, New York, 1987, pp. 365-372

[sscli] The shared source CLI web page available march 2004 at: http://msdn.microsoft.com/net/sscli/

[Tarjan1] Robert E. Tarjan, *Data Structures and Network Algorithms*, Regional Conference Series in Applied Mathematics, SIAM 1983

[Wilson] Paul R. Wilson, *Uniprocessor Garbage Collection Techniques*, Proc of International Workshop on Memory Management in the Springer-Verlag Lecture Notes in Computer Science series., St. Malo, France, September 1992